PROGRAMMABLE STOCHASTIC PROCESSORS

BY

JOHN M. SARTORI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Assistant Professor Rakesh Kumar, Chair
Professor Naresh Shanbhag
Professor Janak Patel
Professor Andrew B. Kahng, University of California at San Diego
Professor Todd Austin, University of Michigan

ABSTRACT

As traditional approaches for reducing power in microprocessors are being exhausted, extreme power challenges call for unconventional approaches to power reduction. Recent research has shown substantial promise for application-specific stochastic computing, i.e., computing that exploits application error tolerance to enable careful relaxation of correctness guarantees provided by hardware in order to reduce power. This dissertation explores the feasibility, challenges, and potential benefits of stochastic computing in the context of programmable general purpose processors. Specifically, the dissertation describes design-level techniques that minimize the power of a processor for a non-zero error rate or allow a processor to fail gracefully when operated over a range of non-zero error rates. It presents microarchitectural design principles that allow a processor to trade off reliability and energy more efficiently to minimize energy when exploiting error resilience. It demonstrates the benefit of using compiler optimizations that optimize a binary to enable more energy savings when operating at a non-zero error rate. It also demonstrates significant benefits for a programmable stochastic processor prototype that improves energy efficiency by carefully relaxing correctness and exposing errors in applications running on a commodity processor. This dissertation on programmable stochastic processors conclusively shows that the architecture and design of processors and applications should be approached differently in scenarios where errors are allowed to be exposed from the hardware to higher levels of the compute stack. Significant energy benefits are demonstrated for design-, architecture-, compiler-, and application-level optimizations for general purpose programmable stochastic processors.

This dissertation is dedicated to my advisor, Rakesh Kumar, in gratitude for his guidance, integrity, humility, thoughtfulness, drive, character, and friendship.

ACKNOWLEDGMENTS

Sincere thanks to Seokhyeong Kang, for the long hours spent working together, Andrew B. Kahng, for demanding excellence, Janak Patel, for thoughtful advice on new ideas, Naresh Shanbhag, for insightful feedback on research, Todd Austin, for dedicating time to serve on my committee, my labmates, for meaningful discussions and solidarity through both trials and fun, and to my parents, for their unfailing prayer, wisdom, and love.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The primary driver for innovations in computer systems has been the phenomenal scalability of the semiconductor manufacturing process, governed by Moore's law, that has allowed us to literally print circuits and systems growing at exponential capacities for the last three decades. The resulting exponentially reducing cost per function has resulted in an unprecedented penetration of technology in homes and beyond, leading to profound impacts on society and quality of life.

Moore's law has come under threat, however, due to the resulting exponentially deteriorating effects of material properties on chip reliability and power. As transistors become smaller (the oxide in a 22 nm process is only five atomic layers thick, and gate length is only 42 atoms across), it is becoming increasingly expensive for the current design and manufacturing technology to keep transistors functioning deterministically, even under normal operating conditions. There are three primary sources of non-determinism [1]. First, decreasing transistor sizes lead to different transistors being doped differently during the manufacturing process, causing them to have non-deterministic electrical characteristics [2]. Second, transistors have become smaller than the wavelength of the light used to pattern them (by more than 6×) [3]. This causes non-determinism in the dimensions and characteristics of the manufactured transistors. Finally, the unprecedented increase in the power density of chips, coupled with time and context-dependent variation in temperature and utilization across the chip, cause voltage and timing variations in circuits [4]. These variations are dynamic and largely non-deterministic. The most immediate impact of such non-determinism is decreased chip yields. A growing number of parts are thrown away, since they do not meet timing and power-related specifications. A 5% yield loss on a 90 nm process today directly translates into a cost to the manufacturer that exceeds 2× the design cost for a typical cellphone manufacturer [5], arguably one of the highest

volume parts. Clearly the status quo cannot continue. Left unaddressed, the entire computing and information technology industry will soon face the prospect of parts that neither scale in capability nor cost. We must find a solution to the non-determinism problem if semiconductor technology and industry are to remain a viable driver of scientific innovation and technological capabilities for the future.

Paradoxically, the problem is not non-determinism, *per se*, but how computer system designers approach it. Chip components no longer behave like the precisely chiseled machines of the past; yet, the basic approach to designing and operating computing machines has remained unchanged. While there have been many swings in computing platform paradigms, such as from general purpose to specialized, and from single-core to multi-core, the contract between hardware and software has remained unchanged. This contract guarantees that hardware will return correct values for every computation, under all conditions. In other words, we demand hardware to be overdesigned to meet the mindsets in computer systems and software design of the past. Guardbands imposed to fake determinism on non-deterministic hardware result in increased cost [6], because getting the last bit of performance incurs too much area and power overhead, especially if performance is to be optimized for all possible computations. Conservative guardbands also leave enormous performance and energy potential untapped, since the software assumes lower performance than what a majority of instances of that platform may be capable of attaining most of the time. As we enter an era where power and performance are first-order design concerns, the cost of faking determinism could be prohibitive, and we may want to revisit the traditional hardware-software contract.

There is another reason why we may want to revisit the hardware-software contract. Several classes of applications can tolerate errors. However, we still design our processors for perfection, as if no application could tolerate any errors. Thus, the traditional compute stack is really overdesigned for many applications that could actually tolerate errors. A more efficient system would perhaps be one in which the reliability of hardware is matched to the reliability requirements of software.

Keeping in mind the non-determinism of hardware and the error tolerance of software, this dissertation examines the feasibility of a computing stack where hardware is allowed to expose errors to software. Specifically, it ex-

plores programmable processors that are designed and architected from the ground up to allow errors under nominal conditions. These processors are called *programmable stochastic processors*. The number and nature of errors that the hardware can expose is determined by the error tolerance available in software or hardware, and error distribution statistics are used to determine how to optimize the processors and applications to maximize energy savings for a given distribution of errors that can be tolerated. This dissertation discusses efforts toward understanding the feasibility, challenges, and potential benefits of building general purpose programmable processors that expose errors to applications and are optimized not for correct operation but for non-zero error rates.

The dissertation is organized as follows. Chapter 2 provides background information, beginning with the foundations of stochastic computing. Chapter 3 introduces a simple implementation of a programmable stochastic processor and presents benefits for an example multimedia application. Chapters 4, 5, and 6, respectively, describe design-, architecture-, and compiler-level optimization techniques for general purpose programmable stochastic processors that are optimized for non-zero error rates. Chapter 7 presents a stochastic processor prototype that demonstrates energy benefits for applications running on a commodity processor. Finally, Chapter 8 summarizes the work and outlines some future directions of research.

CHAPTER 2

BACKGROUND

This chapter provides background information on stochastic computing, beginning with the foundational underpinnings and subsequently describing how the notion of stochastic computing has developed.

2.1 Early Foundations of Stochastic Computing

The foundations of stochastic computing were laid decades ago by computing pioneers such as John von Neumann. Von Neumann believed that proper handling of errors would be achieved not by treating errors as accidental occurrences, but as an essential and meaningful part of computation [7]. He also believed that existing solutions to manage errors through redundancy were unsatisfactory and ad hoc, and desired to see errors treated with theoretical rigor, similarly to the way Shannon had approached the domain of information theory [8]. This dissertation adopts a similar view of stochastic computing as computation on inherently unreliable hardware, where errors are treated as first-class citizens, to be expected in the common case and accounted for during system optimization.

In a seminal work on stochastic computing [7], von Neumann developed the beginnings of a theoretical framework, proving that reliable systems could be synthesized from unreliable components (he considered 3-input majority gates), granted that the component probability of failure is bounded. Conversely, he also proved that for such logics it is impossible to build reliable systems if component reliability is below a certain threshold (1/6 for his formulation using 3-input majority gates [8]). Von Neumann also proposed a formulation of probabilistic computing in which variables are represented by N-bit bitstrings, where the value of a variable is encoded as the probability that a bit in the string is '1'. While von Neumann's work laid the groundwork

for stochastic computing theory, it is worth noting that these initial formulations had significant limitations. For example, even for low component error rates, a huge number of bits (N) are needed to synthesize a reliable system.

Based on the theory that developed as a result of early work in the field, several early stochastic processors were built [9], focusing on special purpose arrays of stochastic computing elements, including machines built at Illinois [10]. Theoretical works also built on the work of von Neumann, producing formulations and bounds for systems comprised of different fundamental components [11, 12, 13], and components with larger error rates [14]. A major drawback of these works is that the reliability of the systems synthesized from unreliable components is substantially lower than the component reliability unless large, complex, redundant networks of components are employed. Also the proposed systems unrealistically assume that component failures are completely independent – a nearly impossible scenario to reproduce in the real world.

The early formulations of stochastic computing primarily focus on providing reliability for systems built from inherently unreliable components. Later work, however, takes stochastic computing a step further. While modern computing systems are indeed synthesized from inherently and increasingly unreliable devices, and our goal is to perform acceptable computation on such systems, not all applications require perfect hardware to achieve acceptable output quality. *Application-specific stochastic computing* techniques exploit error tolerance in applications to carefully relax hardware correctness, especially when hardware correctness is expensive to guarantee. As such, in addition to ensuring acceptable computation on unreliable hardware, application-specific stochastic computing may also provide significant energy benefits.

2.2 Application-Specific Stochastic Computing

One of the early works on application-specific stochastic computing proposes algorithmic noise tolerance (ANT) [15]. ANT proposes the use of low-energy soft digital signal processing (DSP). A soft DSP design uses voltage overscaling or better-than-worst-case design to reduce energy consumption at the expense of some timing errors. Faced with these errors, ANT uses knowledge

of an application's expected output signal distributions to qualify results and filter out outliers. Specifically, ANT employs a reduced complexity estimator block in parallel with the main circuit block. The estimator block computes a reduced-precision, but error-free, version of the output, which is compared against the output of the main block. A large difference between the outputs indicates that a timing error has occurred in the main block. In the case that an outlier is detected, ANT discards the erroneous main block output and uses instead the reduced-precision version of the output. By reducing voltage and mitigating the effect of erroneous computations on output quality, ANT enables reduced energy consumption while minimizing degradation in output SNR.

Following the statistically rigorous communication-inspired design style of ANT, a stochastic sensor network-on-a-chip [16] (SSNOC) uses robust estimation techniques to tolerate errors induced by process variation and voltage overscaling. Due to localized sources of variation, such as particle strikes, thermal hotspots, and process variations, a single, centralized computation resource may be vulnerable to static and dynamic non-idealities. To overcome this vulnerability, an SSNOC architecture decomposes a centralized computation resource into a network of statistically similar [16] sensors. Statistical similarity implies that although individual sensor reading may contain errors, the average value of each sensor equals the expected value of the original computation resource. To reduce implementation cost, the distributed sensors are designed to have reduced complexity. The outputs of the sensors are fused together by a fusion circuit block to produce the final output, using principles of robust estimation theory.

Two sources of errors affect SSNOC computations – estimation errors due to the reduced precision of the distributed sensors and errors induced by process and dynamic variations. Although the mean value of the estimation error is expected to be zero, the distribution of variation-induced errors is unknown. Therefore, there is a random variation component in the final output of the SSNOC.

The fusion block in an SSNOC architecture is responsible for combining sensor outputs to produce the final output of the network, using robust estimation. Depending on the type of computation being performed, the fusion block may take on a different form. This brings to light some potential disadvantages of SSNOC. Namely, the computation must be decomposable into a

distributed network of statistically similar sensors, and the fusion block used to combine sensor data must be custom designed for each SSNOC. In cases where an SSNOC implementation is possible, error detection probability can be significantly improved, due to hardware replication and distribution.

Also following in the communication-inspired vein of ANT and SSNOC is soft N-modular redundancy (NMR) [17]. Like traditional NMR, soft NMR votes between redundant computations, but instead of a majority voter uses a soft voter that employs data and error statistics along with detection theory to determine the most likely correct output. Compared to conventional NMR, soft NMR can significantly reduce system error probability and, in some scenarios, may even be able to produce a correct output when all N redundant computations have errors.

Another early work on application-specific stochastic computing focuses on using relaxed-correctness devices to generate randomness in application-specific circuits that require random inputs with certain probability distributions. Probabilistic CMOS (PCMOS) [18] attempts to exploit the stochasticity of low-energy circuits as a source of randomness in inherently probabilistic applications. PCMOS claims that an inverter operating close to the thermal noise margin will act as a probabilistic bit, with a tunable probability of outputting a '1' or a '0'. Since operating near the thermal noise margin requires a much lower voltage than the nominal voltage, PCMOS logic can produce "random" outputs with lower power consumption. Thus, an application that uses random values should be able to save energy by generating the random values with PCMOS logic.

PCMOS work proposes to create application-specific random value generators, aside from deterministic logic, such that there is a strict partitioning between probabilistic and deterministic components in a PCMOS design. Thus, the probabilistic logic acts as a co-processor that is polled whenever the deterministic components require random values. Because of the strict partitioning that is required, the PCMOS design style may only be suitable for algorithms with well-defined probabilistic steps. Strict partitioning also incurs costs for communication between the deterministic host and the probabilistic co-processor. If the probabilistic step is critical to an application, this communication link may become a bottleneck.

On another note, PCMOS design relies on high quality and controllability of random values produced by probabilistic logic, since the values are inte-

gral to application output quality. It should be noted that the efficiency of PCMOS logic depends on the desired output probability distribution. Also, supporting a wide range of probabilities degrades the efficiency of PCMOS.

Another strand of research extremely relevant to stochastic computing takes advantage of the fact that not all applications or operating conditions exercise the worst-case physical margins of a design. As such, for some applications and operating conditions, energy efficiency can be improved by operating at a better-than-worst-case (BTWC) operating point where hardware correctness is not guaranteed. Relaxing physical design margins and targeting BTWC operation is called timing speculation (TS) [19, 6, 20, 21]. Typically, TS improves energy efficiency by either scaling up the operating frequency (frequency overscaling) or scaling down the operating voltage (voltage overscaling). An overscaled design has higher performance or lower power than its counterpart worst-case design. However, since the delay of some paths may now be greater than the clock period under certain conditions, timing violations can occur, when the correct output of a logic path has not reached the path output in time to be captured in the output register. To account for this occurrence, TS designs often include mechanisms to prevent [22, 23, 24] or detect and correct [6, 19, 20, 21] errors. Since allowing, tolerating, or correcting errors costs performance, power, or output quality, the benefits achieved by TS designs depend on the error rate that overscaling induces.

Razor is a circuit-level TS technique that detects and corrects any timing errors that occur during BTWC operation. It detects timing violations by supplementing critical flip-flops with a shadow latch that strobes the output of a logic stage at a fixed delay after the main flip-flop. If a timing violation occurs, the main flip-flop and shadow latch have different values, signaling the need for correction. Error correction in Razor-based designs involves recovery using the correct value(s) stored in the shadow latch(es). A pipeline restore signal is generated by OR-ing together error signals of individual Razor flip-flops. The signal overwrites the shadow latch data into the errant flip-flop. Recovery mechanisms for Razor-based designs include the use of clock gating and a counter-flow pipeline [25].

Another strand of research relevant to stochastic computing is the work on testing techniques that have been proposed to increase chip yields for specific applications based on the observation that some applications can perform

acceptable computation even when some components of a chip do not work perfectly. Intelligible testing identifies and addresses two potentially limiting characteristics of conventional testing techniques. First, conventional testing techniques classify all chips on a pass-fail basis. There is no range of gradation between these two extremes. Second, conventional testing techniques do not take application characteristics into account and thus cannot rate chips based on how they function for specific target applications. Because classic testing strategies create a strict dichotomy between defect-free and faulty chips, a single hardware fault can cause a chip to be discarded, even if the chip could work acceptably for some applications, regardless of the fault. As we enter the multi-core era (and possibly the dark silicon era), it may become increasingly likely that chips contain circuitry that is not used for all applications. Consequently, scenarios in which faulty chips can function acceptably may also become more prevalent.

For instance, some faults may not affect the behavior of all applications, while other faults may only degrade performance or output quality without causing application failures. If these degradations are tolerable, chips with such faults can still be used. Intelligible testing increases yield by salvaging chips in these two categories. Although these chips have faults, they are either fault-free for some target applications or they have acceptable performance and output quality for the target applications, despite the manifestation of occasional faults.

One limitation of intelligible testing is that it increases test time and cost. Normally, test time is restricted for economic reasons. However, the positive effect on yield due to salvaged chips may be worth the extra testing cost, especially as variability continues to increase with technology scaling.

2.3 Our Work

Application-specific stochastic computing techniques have demonstrated the potential for significant energy benefits from exploiting error tolerance to allow careful relaxation of correctness. However, due to the application-specific nature of error tolerance, benefits have been demonstrated for stochastic computing largely in the context of application-specific circuits (ASICs). Extending stochastic computing to the context of general purpose processors

presents several challenges. For instance, general purpose stochastic processors must adapt their hardware reliability to meet the reliability requirements of different applications with different amounts of error tolerance, such that the error tolerance of each application is exploited for maximum energy reduction. Such processors should be able to make graceful tradeoffs between energy and reliability over a range of non-zero error rates. They should efficiently provide support for applications with multiple phases that have different reliability requirements. They should guarantee safe execution of applications when errors are exposed from hardware to higher levels of the compute stack. Processor hardware should be able to adapt to expose only the number and nature of errors that can be tolerated for a given form of software or hardware error tolerance. Adoption of general purpose stochastic processors requires techniques for bolstering the robustness of several classes of applications to increase the scope of applications that can achieve energy benefits through stochastic computing and to increase the energy benefits available to error-tolerant applications. Widespread adoption also requires the development of automated tools for design and architecture of processors and applications that support a wide range of logic and applications styles. This dissertation explores the feasibility, challenges, and potential benefits of application-aware stochastic computing in the context of general purpose programmable stochastic processors. Specifically, it explores design-, architecture-, compiler-, and application-level optimizations for general purpose programmable stochastic processors.

CHAPTER 3

A PROGRAMMABLE STOCHASTIC PROCESSOR

Future microprocessors will increasingly rely on an unreliable CMOS fabric due to aggressive scaling of voltage and frequency and shrinking design margins. Fortunately, many emerging applications can tolerate computational errors caused by unreliable hardware, at least during certain execution intervals. In this chapter, we present an introductory example of programmable stochastic processors – computing platforms for error-tolerant applications that are able to scale gracefully according to performance demands and power constraints while producing outputs that are, in the worst case, *stochastically correct*. Scalability is achieved by exposing to the application layer multiple functional units that differ in their architecture but share the same functionality. A mobile video encoding application presented here is able to achieve the lowest power consumption at any bitrate demand by dynamically switching between functional unit architectures.

3.1 Introduction

The emergent *ubiquitous computing* paradigm promises new applications in environmental monitoring, automation, and health care. For these new applications to be practical, the computing platform must offer high performance while operating within a very limited power budget (often mW or even nW). While technology scaling driven by Moore's law has offered continued reduction in power consumption and size, recent projections from the ITRS roadmap suggest that this scaling trend alone will not be sufficient to meet the demands of these future applications [1].

Conventional dynamic voltage and frequency scaling techniques [26] are

I would like to acknowledge the contribution of Sriram Narayanan, who worked in collaboration with me to create the mobile video communication application described in this chapter.

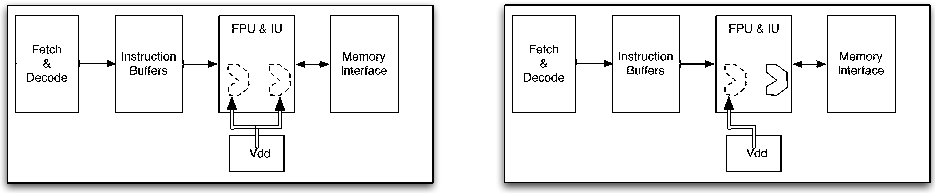
necessarily limited by the particular path delay characteristics of the underlying architecture. In the present day design flow, architectural choices for the various functional units (FUs) are made not with an intent to allow voltage or frequency scaling but to minimize power and area for operation at the nominal voltage and frequency. In traditional high-performance designs, all timing paths are tuned to match the length of the critical path. An implication of this design style is that when one timing path fails, a large number of other timing paths fail, since all path lengths are bunched around the critical path length [27]. This design style prevents effective deployment of hardware-based error tolerance mechanisms [28], suggesting that computational platforms should be designed from the ground up to allow aggressive scaling.

Fortunately, a large class of emerging applications can tolerate a small number of timing errors in computations. Recent research efforts exploit inherent error-tolerance of some applications [29, 30]. While these approaches have been shown to overcome hardware errors, they impose a nontrivial overhead when error rates are low or zero. For instance, if there are execution phases when the system demands the maximum reliability, then it is desirable that the architecture scales to meet these demands.

With the above in mind, we propose *programmable stochastic processors* [31, 32, 33, 34], a computing platform for error-tolerant applications that is able to scale gracefully according to performance demands and power constraints while producing outputs that are, in the worst case, *stochastically correct*. Our proposed architecture gains power savings by exposing to the application multiple functionally equivalent units that exhibit several levels of reliability. This processor *scales* to changing application demands and constraints by dynamically switching between multiple functional units. We choose a video encoding application as an example to showcase the benefits of this design.

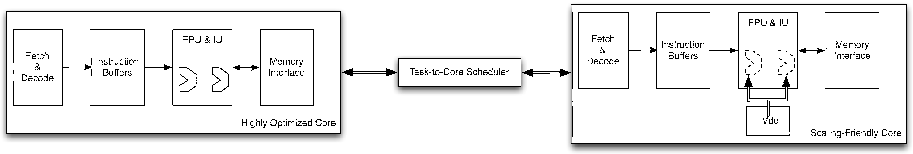
3.2 Soft Architectures

Scalability can be achieved by replacing or supplementing traditional functional units with gracefully degrading units. Such functional units may be incorporated into present day systems at three broad design levels:



(a) Design D1

(b) Design D2



(c) Design D3

Figure 3.1: The scalable architecture introduces alternative functional units at three levels. In (a), all the functional units of the core are replaced by scaling friendly versions; (b) shows two different FU architectures that can be selectively used; and (c) shows a reliability-defined heterogeneous multi-core system.

D1. Fixed: In this design, the baseline architecture is modified by replacing one or more functional units with an alternatively designed functional unit that is more conducive to voltage and frequency scaling. As shown in Figure 3.1(a), the execution unit consists of voltage scaling-friendly blocks (dotted lines) and is able to reduce computation accuracy gradually with supply voltage. This design is suitable for applications that never demand maximum reliability but impose a very limited power budget. This design point represents the least change to existing instruction set architecture (ISA) and programming models, but the ability to scale comes at the cost of compromising best power and performance when such scalability is not desired.

D2. FU selectable: In this design, the baseline processor is equipped with two different functional unit architectures. The application may choose to switch between the two functional units such that the overscaling range is extended. The execution unit in Figure 3.1(b) contains two types of logic blocks – traditional performance optimized version (solid lines) and an alternative design that is friendly to voltage scaling (dotted lines). This design is suitable for applications that have time-varying power or performance demands. We envision a modified ISA that allows the application layers to choose particular functional units. Reliability requirements of applications can be annotated in software, and these annotations can be used to select

the appropriate functional unit for a program or program phase. The current reliability target can be used to control the select lines of a MUX that routes an instruction through the most power-efficient module. Since tuning a module for a specific error rate requires voltage scaling, module switching incurs overhead time for voltage scaling when the module must achieve different reliability targets within the same program.

D3. Core selectable: This design consists of a multi-core system where each core possesses a different architecture for the functional units. Figure 3.1(c) illustrates such a dual-core design along with a task-to-core scheduler that is responsible for assigning tasks according to their reliability requirement. Unlike other multi-core designs such as [30], the scalable cores of this design can dynamically be made error-free by adjusting the supply voltage or clock frequency. This design is suitable for applications that can be decomposed into subcomputations that have different power or performance demands. This design may include design D2 if each core has selectable FU architectures. These cores may or may not share a common ISA. This design is in contrast to systems such as [35] where error-prone cores are avoided or healed; our system exploits them for power savings.

For a class of embedded applications that are data-dominated, it is common for the execution units to significantly contribute to the total power dissipation. For an audio decoding benchmark in the Philips TM3270 media processor [36], the execute module consumes around 0.255 mW/MHz out of a total processor power consumption of 0.935 mW/MHz (a 27% contribution). We restrict our power-reduction design techniques to this class of processors.

3.3 Functional Unit Architectures

To characterize their power and reliability characteristics, functional units are synthesized in the IBM9SF 90 nm CMOS technology with Synopsys DesignCompiler [37], and layout is performed in Cadence SoC Encounter [38]. To measure power and error rate across a range of voltages, we use voltage-specific Synopsys Liberty (.lib) files prepared with Cadence SignalStorm [39]. To obtain an accurate characterization of module behavior, we perform gate-level simulations using an input set of 180k random input samples.

As instances of different functional unit architectures, we consider the

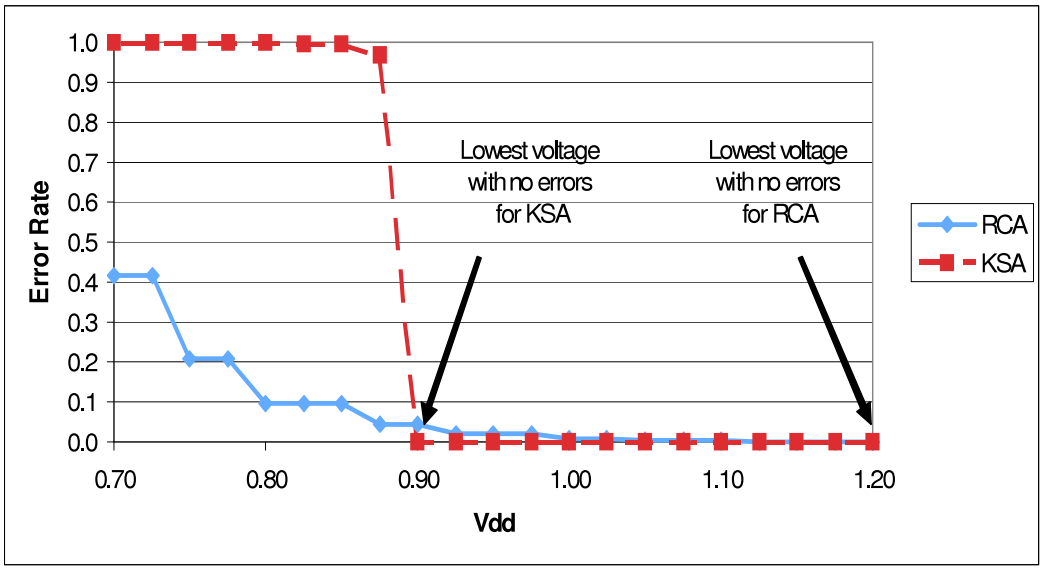


Figure 3.2: The KSA can be scaled to a much lower voltage before reaching the critical point, but fails catastrophically afterward. Errors start to occur for the RCA after the nominal voltage, but they only increase gradually as voltage is scaled down.

Kogge-Stone adder (KSA), which is highly optimized and fails catastrophically with voltage overscaling, and the ripple-carry adder (RCA), which fails gracefully but is much slower than the KSA. Figure 3.2 shows how the error rate of each adder architecture varies as voltage is scaled down.

The KSA can be scaled to a much lower voltage (0.9 V compared to 1.2 V for the RCA) before producing errors. However, once errors occur, the adder fails catastrophically. On the other hand, the error rate of the RCA increases gradually as voltage is scaled down. However, the onset of erroneous behavior is much earlier than in the KSA so that a conservative voltage must be chosen to guarantee fidelity of the output. Because of these failure characteristics, the functionally equivalent modules have very different power-reliability characteristics. Figure 3.3 compares the power consumption of the adders at different error rates.

For reliable operation (0% error rate), the KSA consumes 25% less power than the RCA. This is because for the same frequency, voltage on the KSA can be scaled down to save power, while scaling down voltage on the RCA would cause timing errors. However, power-reliability tradeoffs are not possible for the KSA, since reducing voltage past the critical point causes massive failure, so power consumption is the same for all error rates. While the

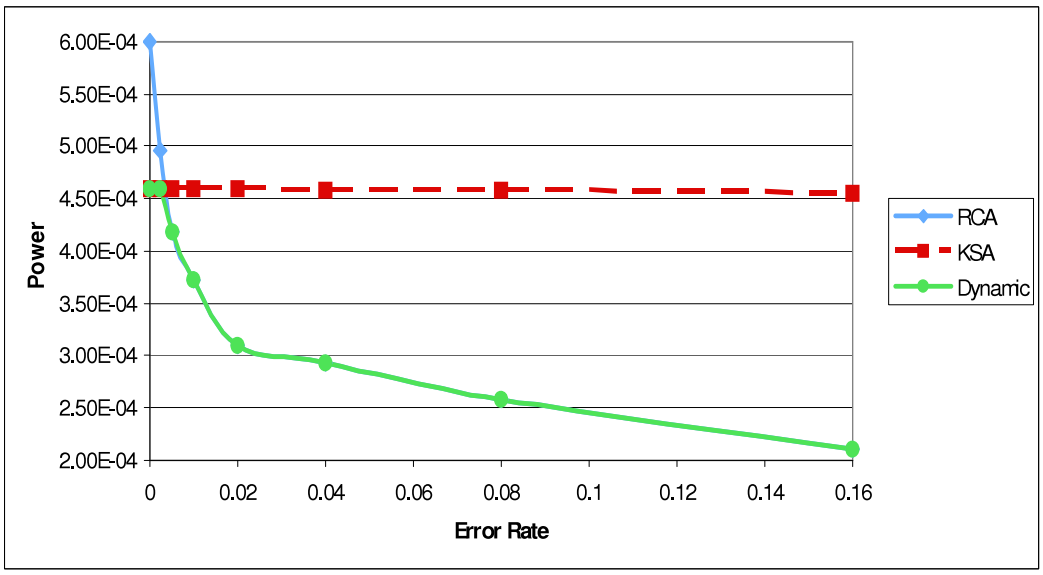


Figure 3.3: The RCA allows power-reliability tradeoffs so that power is reduced as error rate is allowed to increase. The KSA, on the other hand, consumes less power for reliable operation, but does not allow power-reliability tradeoffs.

RCA is less efficient when operating completely reliably, its gradual failure characteristic allows reliability to be traded for power savings, making RCA favorable for noisy environments. For all non-zero error rates, the RCA consumes less power than the highly optimized KSA.

As the error rate increases, gate-level error-recovery mechanisms (e.g., [19]) that exploit gracefully degrading architectures suffer recovery overhead that dominates power savings achieved through voltage scaling. The trend in Figure 3.4 suggests that gate-level techniques that seek to correct every instance of hardware errors may be inefficient in comparison with system-level approaches that do not correct every error instance and allow some errors to be masked. An architecture that allows instructions to be routed to the optimal module for a given system-level error rate can achieve benefits over a static module selection.

3.4 Error-Tolerant Applications

The stochastic processor architecture described above targets aggressive power reduction for a class of error-tolerant applications, i.e., applications that can

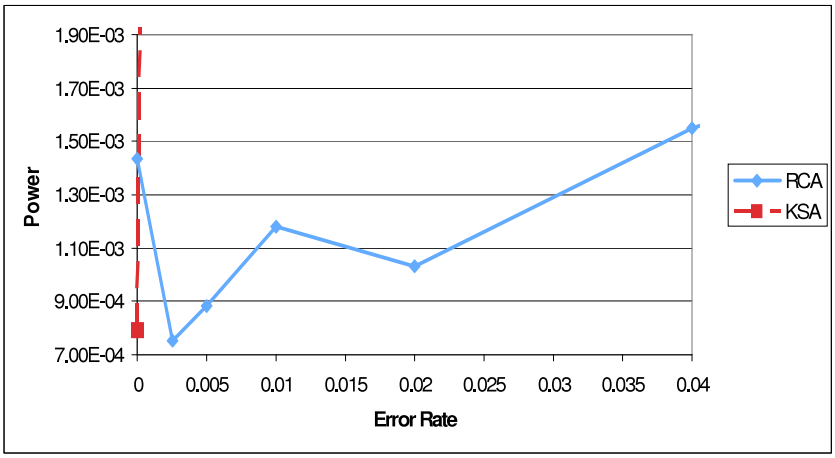


Figure 3.4: Razor error recovery can provide some power savings for gracefully failing designs (RCA) after the point of first error. However, these benefits are limited, since only a small number of errors can be gainfully tolerated before recovery overhead outweighs voltage scaling power reduction. Note that the quantity on the x-axis is the error rate prior to recovery.

operate with *a priori* known reliability requirements. The reliability requirement may change with time and execution phases within the application. Multimedia applications are typical examples of error-tolerant applications. Here, the input (such as a feed from a camera or a microphone) is already contaminated by measurement noise. Since these applications are increasingly implemented on fixed-point mobile platforms, quantization poses another source of noise. Furthermore, the outputs in these applications need only meet the fidelity discernible by human sensory acuity. Programmable stochastic processor architectures can offer significant power and/or throughput gains to these applications, if we treat computational errors as a new source of noise.

As a particular example, we showcase the advantages of a stochastic processor architecture for the popular H.264 video encoding application. The high compression efficiency of this new video encoding standard has enabled exciting applications in wireless video communication and is increasingly implemented on battery-constrained mobile devices. An important subsystem of the video encoder, motion estimation, is often reported as contributing around 40% to 50% of the total encoder power consumption on ASIC implementations [40]. The main computational kernel of the motion estimation

engine is the sum of absolute difference (SAD) that computes $|A-B|$ for two inputs A and B. We seek to gain power savings through voltage overscaling while allowing any resulting timing errors. A computational error in the motion estimation engine simply results in poorer encoding efficiency (i.e., a larger bitrate). Such errors could result in non-zero motion vectors even if the current and reference frames are identical (i.e., there was no motion in the video sequence), which would adversely impact the power overhead of wireless communication. Consequently, during periods of relative inactivity in the input video sequence or favorable wireless channel conditions, the motion estimation block may contain some slack that can be exploited for power reduction. Since models describing wireless communication overhead are beyond the scope of this chapter, we will use the bitrate as a measure of performance. By controlling the occurrence of timing errors in motion estimation, a stochastic processor can trade bitrate for power reduction. But for this approach to work, the bitrate must worsen gradually with scaling supply voltage or clock frequency.

To study the impact of voltage overscaling on compression efficiency of the H.264 video encoder, we used a PC implementation of the JM reference software [41]. The experimental setup described in Section 3.3 was used to obtain the probabilities of bit error for a 16-bit word length. This probability model was used to inject errors into the motion estimation block of the JM reference software. Similar probabilistic models for bit errors caused due to voltage overscaling have also been developed by other researchers [42].

We used three frames of a quarter common intermediate format (QCIF) video source in 4:2:0 YUV format as our input. If there is demand for the lowest achievable bitrate, the application chooses the KSA and is able to consume around 20% less power (when compared with the lowest power option for the RSA that offers this best bitrate). If the application is able to tolerate some worsening of the bitrate, then it switches to the RCA. A small increase in bitrate of around 12 kbits/s (i.e., a 1.2% loss) is able to reduce the power consumption by around 60%. The stochastic processor architecture that is able to switch between the FU architectures at runtime is able to maintain optimal power consumption at all levels of bitrate demand, as shown in Figure 3.5.

As an example implementation, consider design D2. The stochastic processor will receive input from the wireless subsystem (responsible for packetizing

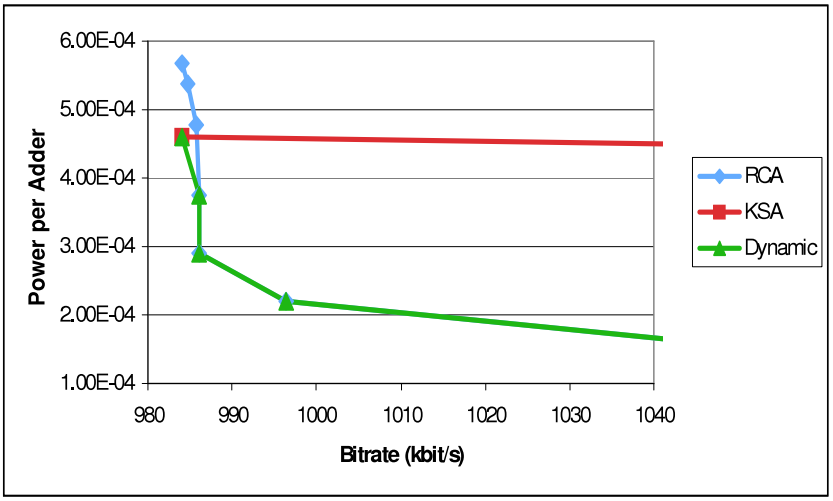


Figure 3.5: The RCA is able to significantly lower the power consumed (per adder) without compromising bitrate of the output. But the KSA is able to offer around 20% lower power consumption when no bitrate degradation can be allowed.

and communicating encoded video data) regarding the quality of the communication channel. Under adverse channel quality conditions, the processor will use the KSA adders by issuing the correspondingly annotated instructions. Under more favorable channel conditions, it will issue the instructions annotated to use the RCA adders. The processor will then proceed to lower the supply voltage according to channel information received from the wireless subsystem. By constantly adapting the issued instructions to changing wireless channel quality, this stochastic processor architecture maintains lowest possible power consumption.

3.5 Summary

Many emerging applications can tolerate occasional computational errors, at least during some execution phases. The stochastic processor architectures presented in this chapter expose multiple alternative functional units to the application and thereby allow more favorable voltage-reliability tradeoffs over a wide range of voltages. This scalability affords 20% to 60% power savings in the motion estimation block of a mobile video communication application.

CHAPTER 4

DESIGN-LEVEL OPTIMIZATION OF PROGRAMMABLE STOCHASTIC PROCESSORS

The example processor in the previous chapter served to introduce the concept and potential benefits of programmable stochastic processors. However, the example was limited in that only a specific class of error-tolerant applications was targeted, and energy-reliability tradeoffs were performed only by routing instructions to different functional units. Subsequent chapters will discuss design-, architecture-, compiler-, and application-level optimization approaches for general purpose programmable stochastic processors. This chapter discusses design-level optimizations that minimize the energy of a processor for a non-zero error rate or allow a processor to make smooth energy-reliability tradeoffs over a range of non-zero error rates.

Conventional CAD methodologies optimize a processor module for correct operation and prohibit timing violations during nominal operation. We propose *recovery-driven design*, a design approach that optimizes a processor module for a target timing error rate instead of correct operation. The target error rate is chosen based on how many errors can be gainfully tolerated by a hardware or software error resilience mechanism. We show that significant power benefits are possible from a recovery-driven design approach that deliberately allows errors caused by voltage overscaling ([15],[19]) to occur during nominal operation, while relying on an error resilience technique to tolerate these errors. We present a detailed evaluation and analysis of such a CAD methodology that minimizes the power of a processor module for a target error rate. We show how this design-level methodology can be extended to design *recovery-driven processors* – processors that are optimized to take advantage of hardware or software error resilience. We also discuss a *gradual slack* recovery-driven design approach that optimizes for a range of error rates to create *soft processors* – processors that have graceful failure charac-

I would like to acknowledge the contribution of Seokhyeong Kang, who worked in collaboration with me to develop the CAD flows described in this chapter.

teristics and the ability to trade throughput or output quality for additional energy savings over a range of error rates. We demonstrate significant power benefits over conventional design – 11.8% on average over all modules and error rate targets, and up to 29.1% for individual modules. Processor-level benefits were 19.0%, on average. Benefits increase when recovery-driven design is coupled with an error resilience mechanism or when the number of available voltage domains increases.

4.1 Introduction

Conventional hardware is designed and optimized using techniques that aim to ensure correct operation of the hardware under different conditions. Conservative design techniques are aimed at ensuring correct hardware operation under worst-case conditions. Better-than-worst-case design techniques [43] save power by eliminating guardbands, but are still aimed at ensuring correct hardware operation under nominal conditions.

In this research, we ask the following question: *Should the availability of an error resilience mechanism change the way we approach hardware design and optimization?* That is, given that mechanisms exist to tolerate hardware errors, should hardware continue to be designed for correct operation or should it be optimized for a target error rate even during nominal operation? To address this question, we propose and evaluate a novel approach to hardware design, called recovery-driven design. Rather than optimizing for correct operation, a recovery-driven design deliberately allows timing errors ([15],[19]) to occur during nominal operation, while relying on an error resilience mechanism to tolerate these errors. In other words, a recovery-driven design optimizes a circuit for a non-zero target error rate that can be gainfully tolerated by hardware [19] or software-based [15] error resilience. The expectation behind recovery-driven design is that the "underdesigned" hardware will have significantly lower power or higher performance than hardware optimized for correct operation. Also, because errors are now allowed, the design methodology can exploit workload-specific information (e.g, activity of timing paths, architecture-level criticality of timing errors, etc.) to further maximize the power and performance benefits of underdesign.

In this chapter, we show that optimizing power for a target timing error

rate for voltage overscaling-induced errors indeed results in significant power savings for similar levels of performance. We show that this is true when errors are detected and corrected by a hardware error tolerance mechanism [19] or allowed to propagate to an error-tolerant application [18] where the errors manifest themselves as reduced performance or output quality [15]. Increasing the target error rate for a processor module increases the potential for power savings, since the module can be operated at a lower voltage. In practice, the target error rate is chosen such that an error recovery mechanism can correct the resulting errors and still reduce energy (after considering the error recovery overhead) for an acceptable degradation in performance or output quality. The power benefits of exploiting error resilience are maximized by redistributing timing slack from paths that cause very few errors to frequently exercised paths that have the potential to cause many errors. This reduces the error rate at a given voltage, and hence reduces the minimum supply voltage and power for a target error rate.

This chapter presents a detailed evaluation and analysis of a recovery-driven design methodology that minimizes the power of a processor module for a target error rate by performing slack redistribution. Our cell sizing-based design-level methodology has been extended to create recovery-driven processors that are optimized for different target error rates or error resilience mechanisms. Since some error resilience mechanisms (e.g., error-tolerant applications) require adaptation to multiple reliability targets, we have also extended our recovery-driven design approach to create *gradual slack* designs – designs that are optimized not for a single error rate, but instead, for a range of error rates. Such gradual slack designs (or *soft processors*) have the ability to trade performance or output quality for energy savings over a range of reliability targets. We make the following contributions in this chapter.

- To the best of our knowledge, we present the first design flow for power minimization that deliberately allows errors under nominal conditions. We demonstrate that such a design flow can result in power savings of 11.8%, on average over all modules and error rate targets, and up to 29.1% for individual modules.

- We explore the heuristic choices and tradeoffs that are fundamental to the optimization quality of slack redistribution-based, recovery-driven

designs. We evaluate choices for path priority and traversal during optimization, optimization radius, accuracy of path selection, error budget utilization, starting netlist, voltage step size granularity, and iterative optimization in terms of their effects on the optimization result, heuristic runtime, and sensitivity to target error rate.

- To support the proposed recovery-driven design flow, we present a fast and accurate technique for post-layout activity and error rate estimation. We use collected functional information to redistribute slack efficiently in a circuit and significantly extend the range of voltage scaling for a target error rate.

- We extend our recovery-driven design methodology to create *recovery-driven processors* (processors that are optimized for different target error rates or error recovery mechanisms) and *soft processors* (processors that are optimized for efficiency over a range of target error rates). We demonstrate the power and energy benefits of such processor designs.

- We demonstrate that the power benefits of recovery-driven processors and soft processors increase when a hardware or software-based error resilience mechanism is used. We consider Razor [19] and application-level noise tolerance [44] as examples and show additional energy reductions of 19% and 20% with respect to the best correctness-optimized processors that exploit the same error resilience mechanisms.

4.1.1 Understanding How Slack and Activity Distributions Determine Error Rate

Before exploring how design-level optimizations affect the efficiency of programmable stochastic processors, we first provide details about our fault model and how slack and activity determine the error rate. The extent of energy benefits gained from exploiting timing error resilience depends on the error rate of a processor. In the context of voltage overscaling-based timing speculation, for example, benefits depend on how the error rate changes as voltage decreases. Likewise, in the context of frequency overscaling, benefits depend on how the error rate changes as frequency increases. If the error

rate increases steeply, only meager benefits are possible [32]. If the error rate increases gradually, greater benefits are possible. In this dissertation, voltage overscaling-based timing speculation is used as a proxy for variation-induced errors. Nonetheless, conclusions should be applicable for other sources of timing variation as well. Note that voltage overscaling may affect reliability in other contexts (e.g., susceptibility to soft errors); however, energy efficiency analysis that considers reliability in other contexts is left as a subject of future work.

The timing error rate of a processor in the context of voltage overscaling depends on the timing slack and activity distributions of the paths of the processor. Figure 4.1 shows an example slack distribution. The slack distribution is a histogram that shows the number of paths in a design at each value of timing slack. As voltage scales down, path delay increases, and path slack decreases. The slack distribution shows how many paths can potentially cause errors because they have negative slack (shaded region). Negative slack means that path delay is longer than the clock period.

From the slack distribution, it is clear which paths can cause errors (timing violations) at a given voltage and frequency. In order to determine the error rate of a processor, however, the activity of the negative slack paths must be known. A negative slack path causes a timing error when it toggles. Therefore, knowing the cardinality of the set of cycles in which any negative slack path toggles reveals the number of cycles in which a timing error occurs.

For example, consider the circuit in Figure 4.2 consisting of two timing paths. P_1 toggles in cycles 2 and 4, and P_2 toggles in cycles 4 and 7. At voltage V_1, P_1 is at critical slack, and P_2 has 3 ns of timing slack. Scaling down the voltage to V_2 causes P_1 to have negative slack. Since P_1 toggles in 2 of 10 cycles, the error rate of the circuit is 20%. At V_3, the negative slack paths (now P_1 and P_2) toggle in 3 of 10 cycles (cycles 2,4,7), and the error rate is 30%.

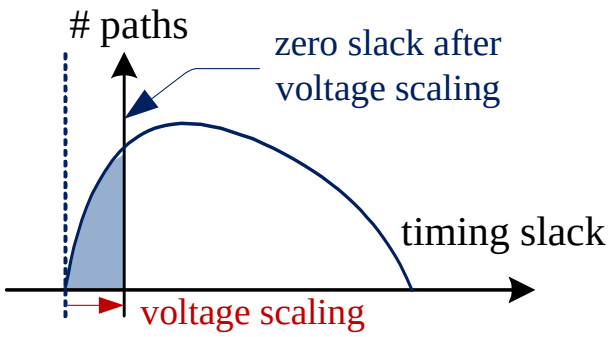


Figure 4.1: Voltage scaling shifts the point of critical slack. Paths in the shaded region have negative slack and cause errors when toggled.

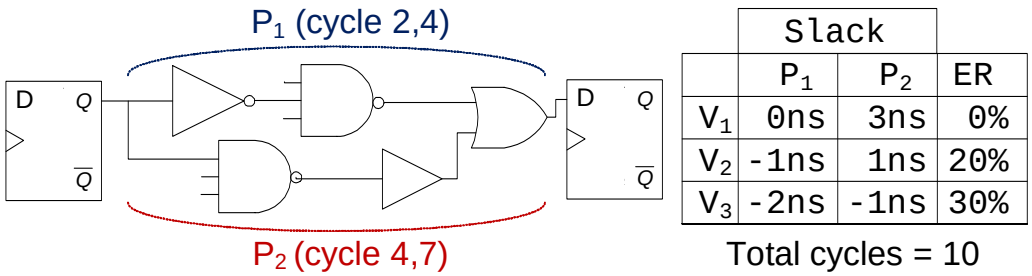


Figure 4.2: Slack and activity distributions determine the error rate.

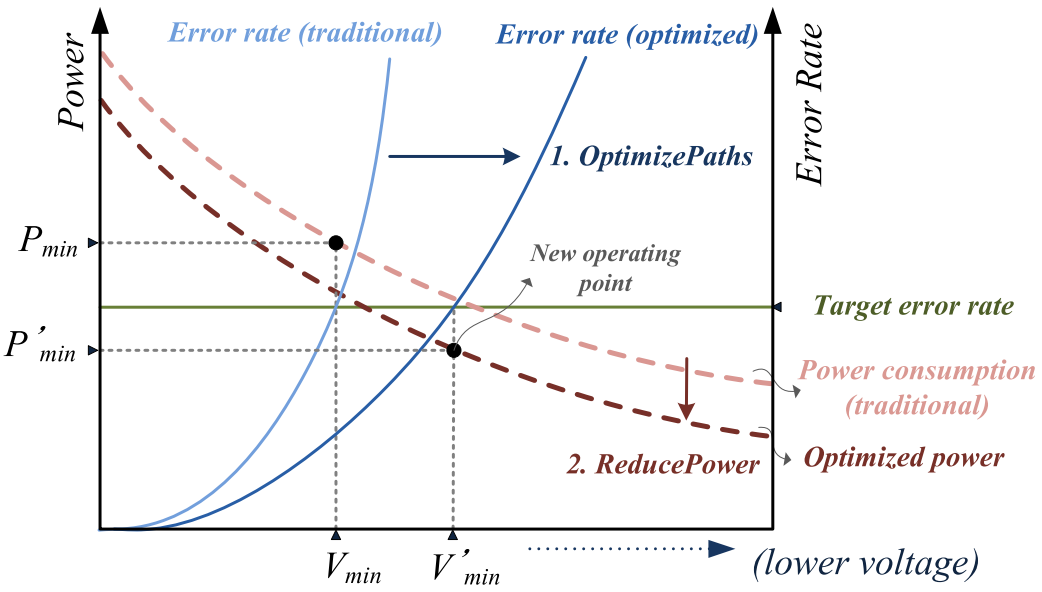


Figure 4.3: Our recovery-driven design optimization redistributes slack from infrequently exercised paths to frequently exercised paths and performs cell downsizing for average-case conditions. These optimizations reduce the power consumption of a circuit and extend the range that voltage can be scaled before a target error rate is exceeded. The combination of these factors produces a design with significantly reduced power consumption.

4.2 Heuristic Design

4.2.1 Motivation

The goal of recovery-driven design in context of voltage overscaling can be stated formally as follows. Given an initial netlist N_0, a set of cell libraries characterized for allowable operating voltages, toggle rates for the toggled paths in the netlist, and a target error rate ER_{target}, produce the optimized netlist $N_{V_{opt}}$ and operating voltage V_{opt} that minimize the total power consumption $W_{V_{opt}}$ of the circuit, such that the error rate of the optimized netlist does not exceed ER_{target}. Figure 4.3 demonstrates the goal.

In this chapter, we present a cell sizing-based design methodology that relies on efficient redistribution of timing slack from infrequently exercised critical paths to frequently exercised paths to reduce the error rate at a given voltage, allowing a reduction in voltage for a given target error rate.

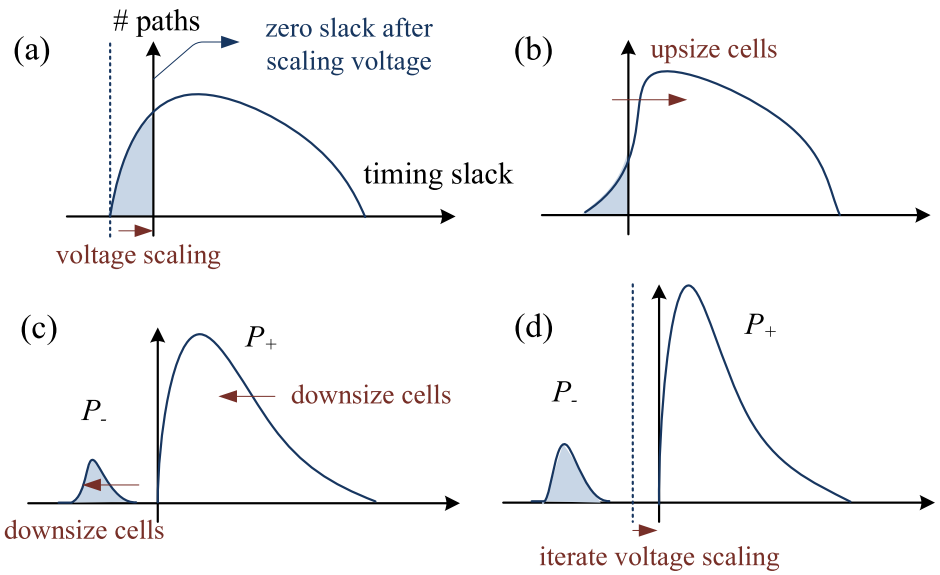


Figure 4.4: The power minimization heuristic reshapes the path slack distribution by redistributing slack from paths that rarely toggle to paths that toggle frequently.

4.2.2 An Abstract Heuristic for Power Minimization

Our heuristic for slack redistribution-based power minimization uses a two-pronged approach – extended voltage scaling through cell upsizing on critical and frequently exercised circuit paths (*OptimizePaths*), and leakage power reduction achieved by downsizing cells in non-critical and infrequently exercised paths (*ReducePower*). The heuristic searches for the combination of the two techniques that results in the lowest total power consumption for the circuit, by performing path optimization and power reduction at each voltage step and then choosing the operating power at which minimum power is observed.

Figure 4.4 illustrates the evolution of the circuit path slack distribution throughout the stages of the power minimization procedure. Each iteration begins as voltage is scaled down by one step (a). After partitioning the paths into sets containing positive and negative slack paths, *OptimizePaths* attempts to reduce the error rate by increasing timing slack on negative slack paths (b). Next, the heuristic allocates the error rate budget by selecting paths to be added to the set of negative slack paths, and downsizes cells to achieve area and power reduction (c). This cycle is repeated over the range of voltages to find the minimum power netlist and corresponding voltage (d).

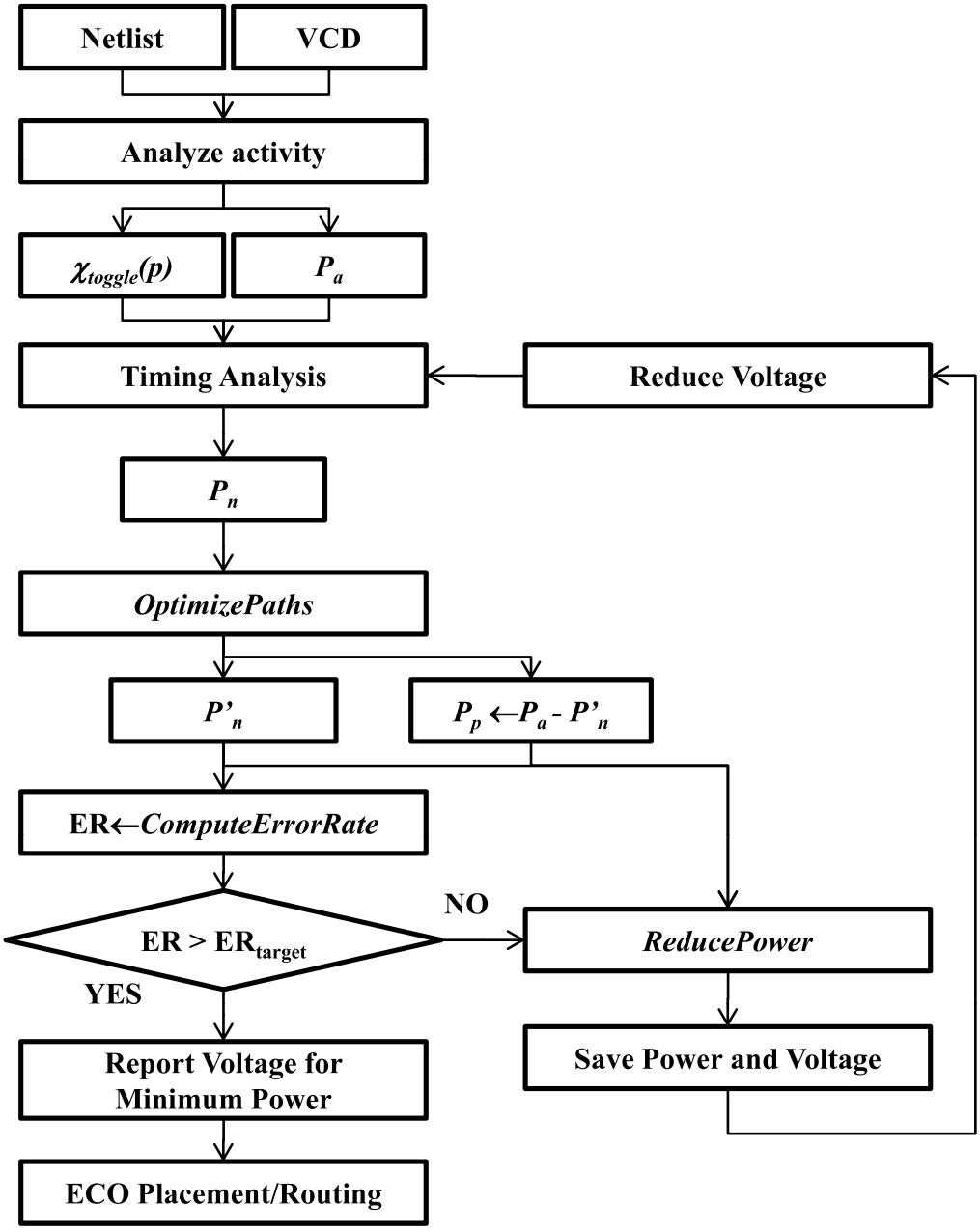


Figure 4.5: Algorithmic flow of a heuristic for minimizing power for a target error rate. P_a is the set of all paths toggled during simulation. P_p is the set of all non-negative slack paths. P_n is the set of all negative slack paths in P_a. $\chi_{toggle}(p)$ is the set of cycles in which path p is toggled.

In Figure 4.4, P_+ is a set of paths that must have non-negative slack after power reduction, and P_- is a set of paths that are allowed to have negative slack. We ensure positive slack for P_+ paths by characterizing timing with worst-case libraries.

Figure 4.5 presents the algorithmic flow of our power minimization heuristic, which couples path optimization to extend the range of voltage scaling (*OptimizePaths*) with area minimization to achieve power reduction (*ReducePower*).

4.2.3 Heuristic Procedures

Path Optimization. The goal of the path optimization procedure (*OptimizePaths*) presented in Algorithm 1 is to minimize the error rate at a voltage level by transforming negative slack paths into non-negative slack paths. This is accomplished by performing cell swaps within the negative slack paths to increase path slack. Negative slack paths with maximum toggle rates are selected first during optimization, since they have the most potential to reduce the error rate if converted into non-negative slack paths.

When a path is targeted for optimization, cell swaps are attempted on all cells in the path to increase slack as much as possible until non-negative path slack is achieved.[1] Once a cell has been visited during optimization, it is marked to prevent degradation of timing slack on any path that the cell is on. Before accepting a cell swap, path slack is checked for all paths that the cell or any visited fanin or fanout cell is on. If the swap caused a decrease in slack for any such path, the move is rejected, and the original cell is restored. Previously optimized (visited) fanin and fanout cells are protected from slack decrease because they belong to paths that have higher toggle rates and, thus, higher priority of optimization. If cell swaps on a path fail to shift the path back into the set of non-negative slack paths, then the path is ignored during subsequent iterations of path optimization.

Any cell swap that increases the error rate (by causing a path to switch from the set of non-negative slack paths to the set of paths allowed to have negative slack) is rejected. Otherwise, we recompute the sensitivity of the swapped cell and all cells in its fanin and fanout networks and select the next

[1]We consider only setup timing slack, since hold violations can typically be fixed by inserting hold buffers in a later step.

cell for downsizing.

Power Reduction. After path optimization, the error rate of the circuit is minimized at the present voltage. From this state, we proceed to minimize the power at the present voltage by utilizing the available error rate budget. Algorithm 1 (*ReducePower*) describes our power reduction procedure. The goal of the power reduction heuristic is to efficiently allocate the remaining error budget to infrequently exercised paths in order to maximize power reduction achieved by cell downsizing. Typically, cells on P_{-} paths can exploit additional downsizing, because these paths are not bound by the normal timing constraint of the circuit.

The first step in power reduction is to choose additional paths to become negative slack paths until the target error rate of the circuit is matched. Paths are selected in order to minimize the additional contribution to the error rate of the circuit. After defining the partition between negative and non-negative slack paths, cell downsizing is performed for all cells in the circuit in order of minimum sensitivity. We define the sensitivity of a cell in Equation 4.1 as the change in cell slack (Δs_c) divided by the change in cell power (Δw_c) when the cell c is downsized by one size. The slack of cell c is defined as the minimum slack on any timing arc containing c. The power of cell c is the sum of static power ($w_{stat}(c)$) and dynamic power ($w_{dyn}(c)$) for the cell. This formulation of sensitivity is similar to those proposed by previous works targeting leakage power reduction [45, 46].

$$Sensitivity(c) = \frac{s_c - s_{c'}}{w_c - w_{c'}} \quad , where \;\; w_c = w_{stat}(c) + w_{dyn}(c) \tag{4.1}$$

4.2.4 Path Extraction and Error Rate Estimation

Path Extraction. Our heuristic has many path-based procedures – *OptimizePaths*, *ReducePower*, and *ComputeErrorRate* – and it is impractical to consider all of the topological paths in these procedures. Therefore, we reduce the number of paths that we consider by extracting only paths toggled during functional simulation. The value change dump (VCD) file can be used to extract toggled paths. To produce a VCD file, we perform gate-level simulation with *Cadence NC-Verilog* [47] at a frequency slow enough to

Algorithm 1 Pseudocode (*OptimizePaths*, *ReducePower*).

Procedure $OptimizePaths(P, N_{V_i}, V_i)$

```
1. Clear 'visited' mark in all cells in the netlist N_{V_i};
2. while P ≠ ∅ do
3.    Select path p from P with maximum toggle rate;
4.    for each cell c in path p do
5.       if c.visited == true then continue;
6.       c.visited ← true;
7.       for each logically equivalent cell m for the cell instance c do
8.          Resize cell c with logically equivalent cell m;
9.          Q ← c ∪ visited fanin and fanout cells of c;
10.         for each path q in P that contains a cell in Q do
11.            if Δslack(q, c, m, V_i) < 0 then restore cell change;
12.         end for
13.      end for
14.   end for
15.   P ← P − p;
16. end while
```

Procedure $ReducePower(P_p, P_n, N_{V_i}, V_i, ER_{target})$

```
1. P_+ ← P_p and P_− ← P_n;
2. while P_+ ≠ ∅ do
3.    Select path p from P_+ with minimum ΔER(p);
4.    ER ← ComputeErrorRate(P_− + p);
5.    if ER ≤ ER_target then
6.       P_− ← P_− + p;    P_+ ← P_+ − p;
7.    else
8.       break;
9.    end if
10. end while
11. Insert all downsizable cells into set C;
12. ComputeSensitivity(C, N_{V_i}, V_i, −1);
13. while C ≠ ∅ do
14.    Downsize cell c from C with minimum Sensitivity(c);
15.    Q ← c ∪ fanin and fanout cells of c;
16.    for each path p in P_+ that contains a cell in Q do
17.       if slack(p, V_i) < 0 then
18.          Restore cell change;
19.          C ← C − c;
20.          continue while loop;
21.       end if
22.    end for
23.    ComputeSensitivity(Q, N_{V_i}, V_i, −1);
24.    if cell c is not downsizable then
25.       C ← C − c;
26.    end if
27. end while
```

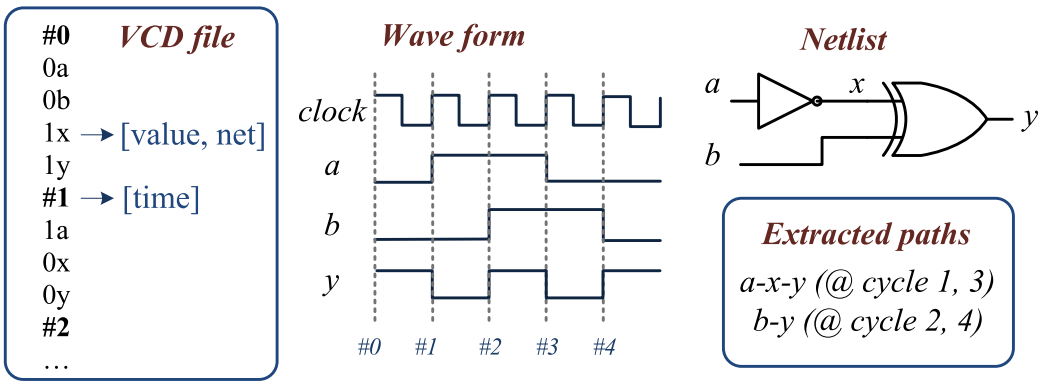


Figure 4.6: VCD file format and path extraction.

capture all possible signal transitions. Figure 4.6 shows an example VCD file and the path extraction method. The VCD file contains a list of toggled nets corresponding to each time at which a transition occurs, as well as their new values. We can use this information to extract toggled paths in each cycle. Nets that glitched or toggled in each cycle are marked, and these nets are traversed to find toggled paths. We detect a toggled path when toggled nets compose a connected path of toggled cells from a primary input or flip-flop input to a primary output or flip-flop output. In Figure 4.6, nets a, x, and y have toggled in the first and third cycles (#1, #3), and nets b and y have toggled in the second and fourth cycles (#2, #4). We extract two paths: $a - x - y$ and $b - y$.

Toggle Rate and Error Rate Estimation. In order to accurately minimize power for a target error rate, we must be able to produce accurate estimates for error rate during our optimization flow. Thus, we propose a novel approach to error rate estimation that enables design for a target error rate.

We calculate the toggle rate of an extracted path using the number of cycles in which the path toggles. $\chi_{toggle}(p)$ represents the set of cycles in which path p has toggled during the simulation. $TR(p)$ represents the toggle rate of path p and is defined as:

$$TR(p) = \frac{|\chi_{toggle}(p)|}{X_{tot}} \tag{4.2}$$

where $|\chi_{toggle}(p)|$ is the number of cycles in which path p has toggled, and X_{tot} is the total number of cycles in the simulation. Using the toggled cycle

information of negative slack paths, we can calculate the error rate precisely. The error rate (ER) of the design is calculated as:

$$ER = \frac{|\bigcup_{p \in P_n} \chi_{toggle}(p)|}{X_{tot}} \tag{4.3}$$

where P_n is the set of negative slack paths in the set of all toggled paths. In Figure 4.6, if paths $a - x - y$ and $b - y$ both have a toggle rate of 0.4 (number of toggled cycles is 2 and number of total cycles is 5), and if path $a - x - y$ has negative slack, then timing errors will occur in cycles #1 and #3. Therefore, the error rate is 0.4 for this example

Our novel technique for error rate estimation has proven to be much faster than functional simulation and more accurate than previous estimation techniques. Results comparing our VCD-based technique to functional simulation and previous estimation approaches can be found in [48].

4.2.5 Heuristic Design Choices

In this section, we discuss heuristic design choices.

Experiment 1: Path Ordering During Optimization. The order in which we select paths for optimization affects the optimization result, since we prevent cells from being visited multiple times during optimization. The order also matters because we protect previously optimized paths from slack degradation due to other attempted cell swaps, as previously optimized paths have a higher optimization priority. We evaluate two prioritization functions for path selection during optimization. The first ranks paths in order of decreasing toggle rate ($TR(p)$). Paths with the highest toggle rates have the greatest potential to decrease error rate when optimized. We compare against a function that ranks paths in order of decreasing $TR(p)/|slack(p)|$. In this alternative, we prefer paths with smaller negative slack, since the least effort is required to convert these paths into non-negative slack paths.

Experiment 2: Optimization Radius. The goal of optimization is to maximize the slack of a targeted path through cell swaps. We evaluate two alternatives for the radius of optimization. In one case, we only swap cells on the target path. In the second case, we target both the cells on the path as well as cells in their fanin and fanout networks, since swaps in the fanin

and fanout networks can also affect cell slack.

Experiment 3: Path Traversal During Optimization. When optimizing a path, the order in which cells are visited can have an effect on the optimization result, since cell swaps affect input slew and output load. We consider two options – traversal from front to back and from back to front. We iterate over the cells in a path and make swaps until there is no further increase in the path slack.

Experiment 4: Accuracy of Path Selection During Power Reduction. During power reduction, non-negative slack paths are selected to be added to the set of paths allowed to have negative slack, thus utilizing the available error rate budget. Paths are prioritized in order of increasing incremental contribution to error rate, $\Delta ER(p)$. However, after moving a path from P_+ to P_-, $\Delta ER(p)$ can change for paths that shared error cycles with the moved path.

To obtain precise ordering in terms of error rate contribution, we can update $\Delta ER(p)$ after each path selection. However, this introduces a runtime overhead, since we must continuously update $\Delta ER(p)$ for all remaining P_+ paths. We compare such *precise prioritization* against the alternative case where $\Delta ER(p)$ is calculated only once for all P_+ paths before path partitioning.

Experiment 5: Error Rate Budget Utilization. During power reduction, the final error rate after cell downsizing could be less than the target error rate, ER_{target}, since some paths in P_- might still have non-negative slack, even after maximum downsizing on the path cells. In this case, we might continue to reduce the power of the design by selecting more paths to add to P_- and downsizing cells again. We evaluate two cases – one where a single pass is performed for path selection and cell downsizing, and one where the *ReducePower* procedure is repeated until there is no further reduction in power (i.e., repeat *ReducePower* whenever some paths added to P_- still have non-negative slack after cell downsizing).

Experiment 6: Starting Netlist. Here, we evaluate heuristic performance for different starting netlists corresponding to loose (clock period increased by 10%) and tight (reduced by 40%) timing constraints. This can significantly affect the final voltage reached, the dependence on engineering

change order (ECO), and the amount of power savings afforded by the power minimization algorithm.

Experiment 7: Voltage Step Size. In each iteration of the power minimization heuristic, we step down the voltage by a value V_{step} and run the *OptimizePaths* and *ReducePower* procedures to produce a netlist for the present level of voltage scaling. The size of V_{step} can influence the optimization result and runtime of the heuristic. Thus, we compare two values of V_{step} – 0.01 V and 0.05 V – and compare the characteristics of the final netlist as well as the heuristic runtime.

Experiment 8: Iterative Optimization. In each iteration of the heuristic, we perform optimization of negative slack paths at that voltage level. During the next iteration, we have a choice between starting from the previously optimized netlist ($N_{V_{i-1}}$) or the original netlist (N_0). We compare the netlists produced in each case to see if they have similar power and runtime characteristics.

4.2.6 Gradual Slack Design

We extend our design methodology to implement another form of recovery-driven design called *gradual slack design* [33], which reshapes the slack distribution of a processor to create a gradual failure characteristic, rather than the typical critical wall. While error rate-optimized, recovery-driven designs achieve better energy efficiency at a single target error rate, gradual slack designs have the ability to trade reliability, throughput, or output quality for energy savings over a range of error rates. Figure 4.7 shows the optimization approach for gradual slack design.

To achieve a gradual slack distribution with our recovery-driven design flow, we do not optimize for a single target error rate by selecting P_{-} paths. Instead, we select the maximum target error rate corresponding to the desired range of scalability, and optimize only the negative slack paths in the scaling range with the highest switching activity, in order to maximize the range of voltage scalability for the target range of error rates. We downsize only cells that have negligible activity so that the slack distribution for the active paths and the error rate of the processor are not affected. In this way, we maintain the desired gradual sloping slack distribution rather than create

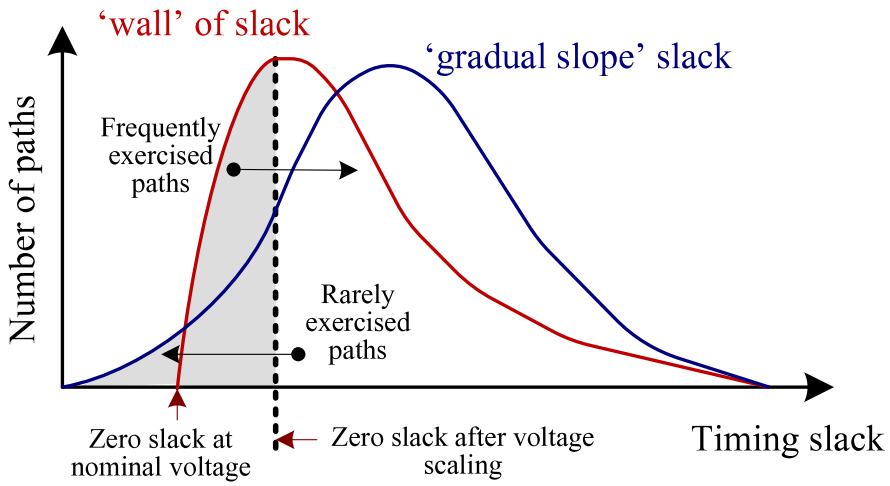


Figure 4.7: The goal of the 'gradual slope' slack optimization is to transform a slack distribution having a critical 'wall' into one with a more gradual failure characteristic. This allows performance-power tradeoffs over a range of error rates.

a critical wall distribution with a cluster of active paths in the permanent negative slack region.

4.2.7 Processor Power Reduction

Algorithm 2 gives a heuristic for minimizing the power of a processor core for a target error rate. The first step of the above power-minimization heuristic involves characterizing the modules of the processor core in terms of their power consumption at different error rate and voltage targets. These data are provided by *PowerOptimizer* and are used to select the optimal operating voltage(s) for the processor core, as well as the error rate targets to assign to the processor modules.

The next step in the processor-level heuristic is to use the data from *PowerOptimizer* to solve an optimization problem. The optimization objective is to minimize the power of the processor core subject to the constraint that the processor error rate must be less than the chosen target rate. Using the data from *PowerOptimizer*, we can formulate expressions for the power and error rate of the processor core in terms of the module error rates and the operating voltage. Thus, the goal of the optimization problem for a particular voltage is to find the assignment of error rate targets to modules

Algorithm 2 Processor-Level Design Heuristic.

Procedure $OptimizeProcessor(ER_{target}, MODULES, DOMAINS)$

1. **for each** module m in the optimization list of MODULES **do**
2. **for each** error rate $ER < ER_{target}$ **do**
3. $PowerOptimizer(N(m), ER)$;
4. **end for**
5. Use the results from $PowerOptimizer$ to characterize $P_m(V, ER)$
6. **end for**
7. **for each** voltage $V \in V_{range}$ **do**
8. Minimize $P_{core}(V) = \Sigma(P_m(V, ER))$ s.t. $ER_{core}(ER_{module_1}, ..., ER_{module_M}) \leq ER_{target}$
9. Record minimum power $P_{core}^{min}(V)$ and module error rate assignment $S(V) = [ER_{module_1}, ..., ER_{module_M}]$
10. **end for**
11. Select the voltage V_{opt} at which power P_{core}^{min} is minimized
12. Let $V^*(S(V)[m])$ be the voltage that minimizes power for module m at $ER = S(V)[m]$
13. Locate the $DOMAINS$ neighbors $\{V_1, ..., V_{DOMAINS}\}$ nearest to the set of voltages $V^*(S(V_{opt}))$
14. Assign each module m to the voltage domain $V_D[m] \in \{V_1, ..., V_{DOMAINS}\}$ that minimizes power $P_m(V_D[m], S(V_{opt})[m])$
15. Layout the processor, selecting for each module $m \in$ MODULES the netlist $N(m, V_D[m], S(V_{opt})[m])$;

that satisfies the optimization objective. We use a disjunctively constrained knapsack-based [49] approach to solve the optimization problem. The knapsack solver selects the voltage and error rate assignment for which the power of the processor core is minimized and uses the selected error rate-optimized netlist of each module to lay out the processor.

For multiple voltage domain designs ($DOMAINS > 1$), the heuristic selects the voltage level of each domain and the partitioning of modules to voltage domains to minimize core power. This process involves first selecting the error rate targets for the modules based on a minimum-power global assignment, then selecting the levels for the voltage domains and module-to-level assignments such that the power of the modules is minimized. The latter step is performed using a nearest neighbor search to identify the neighbors nearest to the set of optimal module voltages corresponding to the module error rate assignments in the space of voltages.

4.3 Recovery-Driven Processors

The proposed design methodology enables *recovery-driven processors* – processors that are optimized to deliberately produce timing errors at a rate that can be gainfully tolerated by an error recovery mechanism. Below, we describe two recovery-driven processor designs – one targeting hardware-based error resilience and another targeting software-based error resilience.

Case Study: Circuit-Level Timing Speculation. One popular hardware-based scheme for error detection and correction is circuit-level timing speculation [19, 50]. Circuit-level timing speculation-based techniques detect errors by sampling the same computation twice – once using the regular clock and again using a delayed clock. The two outputs are compared. When the outputs do not match, an error is signaled. Correction involves treating the delayed clock output as the correct output. Razor [19] and error detection sequential (EDS) circuits [50] provide good examples of circuit-level timing speculation.

A recovery-driven processor design targeted for Razor takes into account the frequency of errors that can be gainfully tolerated by Razor (determined by the dynamic error recovery overhead) as well as the number of latches in which an error may occur (which determines the cost of making the circuit robust to errors). For the design-level heuristic, this means that when we define the partition between paths that are allowed have errors (P_-) and paths that are error-free (P_+), we must consider the error rate contribution of each path, which adds to the dynamic recovery overhead of Razor. We must also account for the cost of using a Razor FF at the endpoint of any path that may potentially cause a timing error, and of buffering for any short paths terminating at that endpoint. If downsizing a path during *ReducePower* requires that we must replace a regular FF with a Razor FF, then we should ensure that the energy benefit (in terms of power reduction for additional cell downsizing) outweighs the additional cost of the Razor FF and any short-path hold buffering. Since Razor assumes a maximum delay constraint on all paths [51], in addition to checking P_+ paths for negative slack (line 16 of *ReducePower*) we must also ensure that all P_- paths respect the delay constraint after a downsizing move.

Case Study: Application Noise Tolerance. Error-tolerant applica-

tions [44] represent an opportunity to save power and increase performance by allowing errors to propagate to the application level rather than expending power to detect and correct them at the hardware level. For several such applications, data errors simply result in reduced output quality, instead of program failure.

Designing a recovery-driven processor for error-tolerant applications requires several considerations. First, the set of processor modules is partitioned into two subsets – one containing modules that produce errors that the applications can tolerate and another containing modules that should not allow errors to propagate to the application level. For the class of error-tolerant applications that we consider in this chapter, errors in the arithmetic units (i.e., ALU, FPU) can be tolerated. For this class of applications (which relies heavily on numerical computation), the arithmetic units account for approximately 35% of the dynamic power consumption of the processor.

In addition to the list of modules to optimize, the *OptimizeProcessor* procedure requires a target error rate. The error rate is chosen such that all applications in the class have acceptable quality for the target error rate.

For the modules that produce errors that the application cannot tolerate, one of two approaches can be followed. One option is to operate those modules on the same voltage rail as the modules in which faults are allowed (single-rail design). In this case, we feed these modules to the optimization heuristic targeting some hardware recovery mechanism that guarantees correctness, such as Razor. The two groups must agree on a common voltage that minimizes power consumption for the entire processor, and the optimal voltage reported by the optimization heuristic can be used as a constraint for the second optimization. Alternatively, the two groups can operate in separate voltage domains (dual-rail design), in which case each optimization can select a different optimal voltage.

Soft processor design can also be used to adapt the reliability of the processor for reliability-diverse workloads, with more power savings available as the error rate target decreases. To create a soft processor design, the gradual slack module-level heuristic is used, and the optimal voltage and error rate targets of the modules are chosen based on the range of error rate targets that the processor should support.

4.4 Methodology

Our methodology for demonstrating the benefits of recovery-driven design has two parts – a design-level methodology to characterize the power and reliability of circuit modules optimized for different voltage and error rate targets, and an architecture-level methodology to estimate processor power and performance when the proposed design-level techniques are applied at the processor-level.

4.4.1 Design-Level Methodology

We use the OpenSPARC T1 processor [52] to test our optimization framework. Table 4.1 describes the selected modules and provides characterization in terms of cell count and area. Module designs are implemented in TSMC 65GP technology using a standard flow of synthesis with Synopsys Design Compiler vY-2006.06-SP5 [37] and place-and-route with Cadence SoC Encounter v8.1 [38]. Runtime is reduced by adopting a restricted library of 66 commonly-used cells[2] (62 combinational and 4 sequential). Conventionally constrained designs are synthesized for the target operating frequency (0.8 GHz), and tightly constrained designs are synthesized for a 40% smaller clock period to increase timing slack.

Figure 4.8 illustrates our recovery-driven design flow. We perform gate-level simulation to produce a VCD file[3] using Cadence NC-Verilog v6.1 [47]. To find timing slack and power values at specific voltages, we prepare Synopsys Liberty (.lib) files for each voltage from 1.00 V to 0.50 V in 0.01 V increments, using Cadence Library Characterizer v9.1 [39]. Complete characterization for 51 voltage points takes a couple of days, but this is a one-time cost.

Timing information is continually available from Synopsys PrimeTime c2009.06 [53] static timing tool through the Tcl socket interface, during the optimization process. After our optimization, all netlist changes are realized

[2]Heuristic efficiency depends on the number of available logically equivalent cells. Since we use all available cell sizes for different drive strengths, our heuristic will also be effective with a full set of library cells.

[3]Gate-level simulation is performed for one million cycles, and the size of the VCD file is about 500 MB for our test cases. To implement larger designs, a compressed VCD file could be used – e.g., Synopsys VCD Plus format.

Table 4.1: Target modules for experiments.

Module	Stage	Description	Cell #	Area (μm^2)
lsu_dctl	MEM	L1 Dcache Control	4537	13850
lsu_qctl1	MEM	LDST Queue Control	2485	7964
lsu_stb_ctl	MEM	ST Buffer Control	854	2453
sparc_exu_ecl	EX	Execution Unit Control	2302	7089
sparc_ifu_dec	FD	Instruction Decode	802	1737
sparc_ifu_errdp	FD	Error Datapath	4184	12972
sparc_ifu_fcl	FD	L1 Icache and PC Control	2431	6457
spu_ctl	SPU	Stream Processing Control	3341	9853
tlu_mmu_ctl	MEM	MMU Control	1701	5113

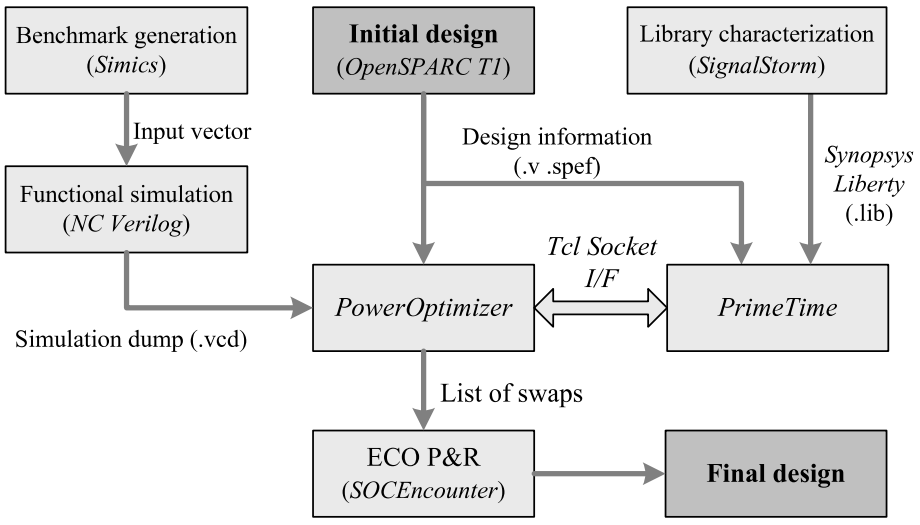


Figure 4.8: CAD flow incorporating the power optimization heuristic to minimize the power of a design for a given error tolerance technique.

Table 4.2: Benchmarks.

Benchmarks for design optimization (training set)	
ART	Image Recognition using Neural Nets
BZIP2	Compression
MCF	Combinatorial Optimization
MESA	3D Graphics Library
Benchmarks for design evaluation (test set)	
EQUAKE	Seismic Wave Propagation
GZIP	Compression
TWOLF	Place and Route Simulator
SORT	Sorting
Additional benchmarks for processor-level evaluation	
AMMP, APPLU, MGRID, PARSER, SWIM, CRAFTY, EON, WUPWISE VPR, VORTEX-2, FACEDETECT†, CG†, LSQ† († error-tolerant application)	

using Cadence SoC Encounter in ECO mode.

Gate-level simulation is performed using test vectors obtained from full-system RTL simulation of a benchmark suite consisting of integer and floating point SPEC benchmarks. These benchmarks are each fast-forwarded to their early SimPoints using the OpenSPARC T1 system simulator, Simics [54] Niagara. After fast-forwarding in Simics, the architectural state is transferred to the OpenSPARC RTL using CMU Transplant [52].

Our recovery-driven design techniques optimize for average activity. To ensure that the activity profiles used during optimization (training) are representative and adequate, we use mutually exclusive training and test workloads. We optimize based on the average activity of half of our benchmarks and test using the other half. Training and test sets are chosen randomly and contain half integer and half floating point benchmarks. Table 4.2 shows the benchmarks in the training and test sets.

When characterizing Razor-based designs, we use worst-case timing libraries to determine any path that might have negative slack under worst-case PVT variations. We assign a Razor FF to the endpoint of any such path, add a maximum delay constraint of 1.5 cycles to the path, and add a minimum delay constraint of 0.5 cycle to all paths ending at that FF. We add buffers to any path that does not meet the minimum delay constraint.

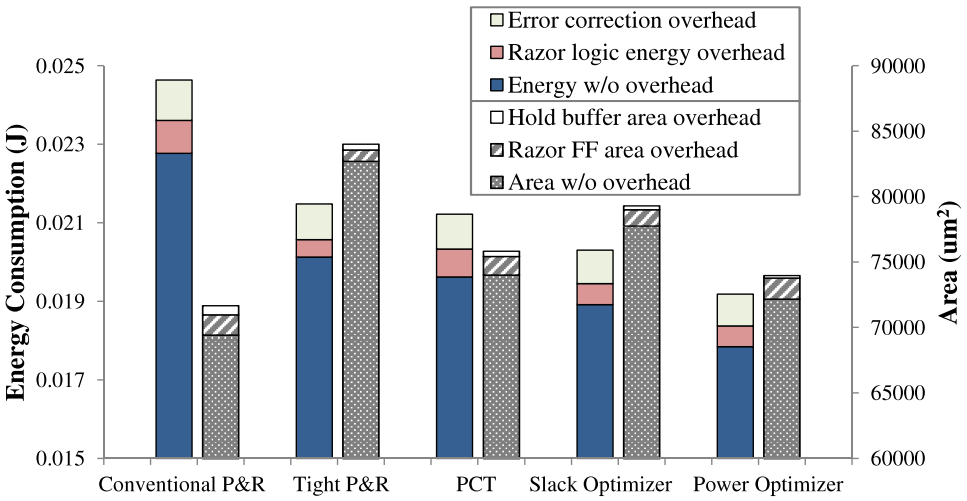


Figure 4.9: Energy and area overheads for Razor-based design.

Razor FFs have higher power, delay, and area than normal FFs [19]. An error triggers a recovery period during which the pipeline recovers to a correct state. During this time, we assume that no progress is made, but we do account for the power and time consumed during recovery when reporting processor throughput and energy. We assume a counterflow pipeline-based Razor implementation [19] with a recovery penalty proportional to the depth of the pipeline (nine cycles for our 9-stage pipeline). We use the error rate, in conjunction with the rates of power consumption during normal operation and error recovery, as well as the recovery time overhead of Razor to calculate the energy overhead of error recovery [19]. Figure 4.9 compares the energy and area overheads of Razor for each design style that we evaluate.

4.4.2 Architecture-Level Methodology

We use SMTSIM [55] integrated with Wattch [56] to simulate processors whose single-core parameters are in Table 4.3. The simulator reports performance and power numbers at different voltages. Our evaluations are done using benchmarks in Table 4.2. These benchmarks were chosen to maximize diversity in terms of performance and reliability requirements. We base our out-of-order processor microarchitecture model on the MIPS R10000 [57].

To get a processor-wide error rate at a given frequency and voltage, we first sum the error rates from all the sampled OpenSPARC modules and

Table 4.3: Processor specifications.

Property	Value	Property	Value
L1 cache	16 kB, 4-way, 1 cyc	RegFile	72 (int), 72 (FP)
L2 cache	2 MB, 8-way, 8 cyc	Branch Predict	gshare (8k entries)
Execution	2-way OO	Mem Access	315 cyc

then scale up the sum based on area, such that it includes all modules that we target for optimization. The error rate of a module that has not been characterized is assumed to be proportional to area. We target only logic modules with our recovery-driven design methodology. On-chip memories are assumed to operate on a separate voltage rail [58] at the lowest error-free voltage for a given operating frequency. At 45 nm and below, such "split rail" designs are common. While we provision for error-free SRAMs, logic interfacing with SRAM structures, such as register read and writeback logic, may still produce errors. For designs that rely on error-tolerant applications, we scale the error rates of each module group separately, according to an error rate characterization of sampled modules in the group. Once the processor core-wide error rate is calculated, we can use performance and power numbers reported by our simulators to estimate the throughput and power impact of errors for a given error recovery overhead.

We use a similar methodology to get processor-wide power numbers. To get a dynamic power estimate, we scale the dynamic power numbers reported by Wattch for the optimizable components by the ratio of total module power for an optimization technique over total module power for the baseline design, as reported by Synopsys PrimeTime. For designs that exploit application-based error resilience, we scale the power of the module groups independently, as we did for error rate. For the non-optimizable components, the Wattch numbers are scaled based on the minimum voltage that these components can run at without producing timing errors. For static power estimation, we use the ratio of dynamic and static module power for an optimization technique, as reported by PrimeTime, to calculate static power for a dynamic power value reported using the above methodology.

When a processor designed for application-level reliability runs an application that requires correctness, we scale down the frequency of the processor

so that no timing violations occur. The safe clock frequency of the design is determined by the worst-case negative slack timing path in the processor plus a safety margin. All of our application simulations are executed for 1 billion cycles after fast-forwarding to the early SimPoints [59].

4.5 Experimental Results

We now evaluate our recovery-driven design implementations, which redistribute timing slack to reduce the error rate at a given voltage, allowing a reduction in voltage and energy for a given target error rate and operating frequency.

4.5.1 Evaluation of Heuristic Design Choices

Figure 4.10 shows power and runtime of the various heuristic design alternatives that we evaluated, as described in Section 4.2.5. For **path ordering during optimization**, considering the slack in the prioritization function results in higher power than the case where only toggle rate is used. Runtime is somewhat smaller, but since our optimization iterates over a path multiple times until no slack increase is observed, both results perform similarly. For the same reason, **path traversal order** has little effect on the optimization result. We choose the toggle rate priority function for its simplicity and lower power.

The results for **optimization radius** show that swapping cells in the fanin and fanout networks not only increases power at some error rates, but also greatly increases runtime due to the large amount of swaps that are performed. Thus, we choose to swap cells only on the optimized path. In the experiments on **accuracy of path selection** and **error rate budget utilization**, we observe no difference in power. Both updating the error rate contribution continuously during path selection and ensuring full utilization of the error rate budget increase runtime significantly without providing power benefits, and these techniques are not used in the final heuristic implementation.

The choices of **starting netlist** and **voltage step size** have a significant effect on power. Our recovery-driven design heuristic employs two main

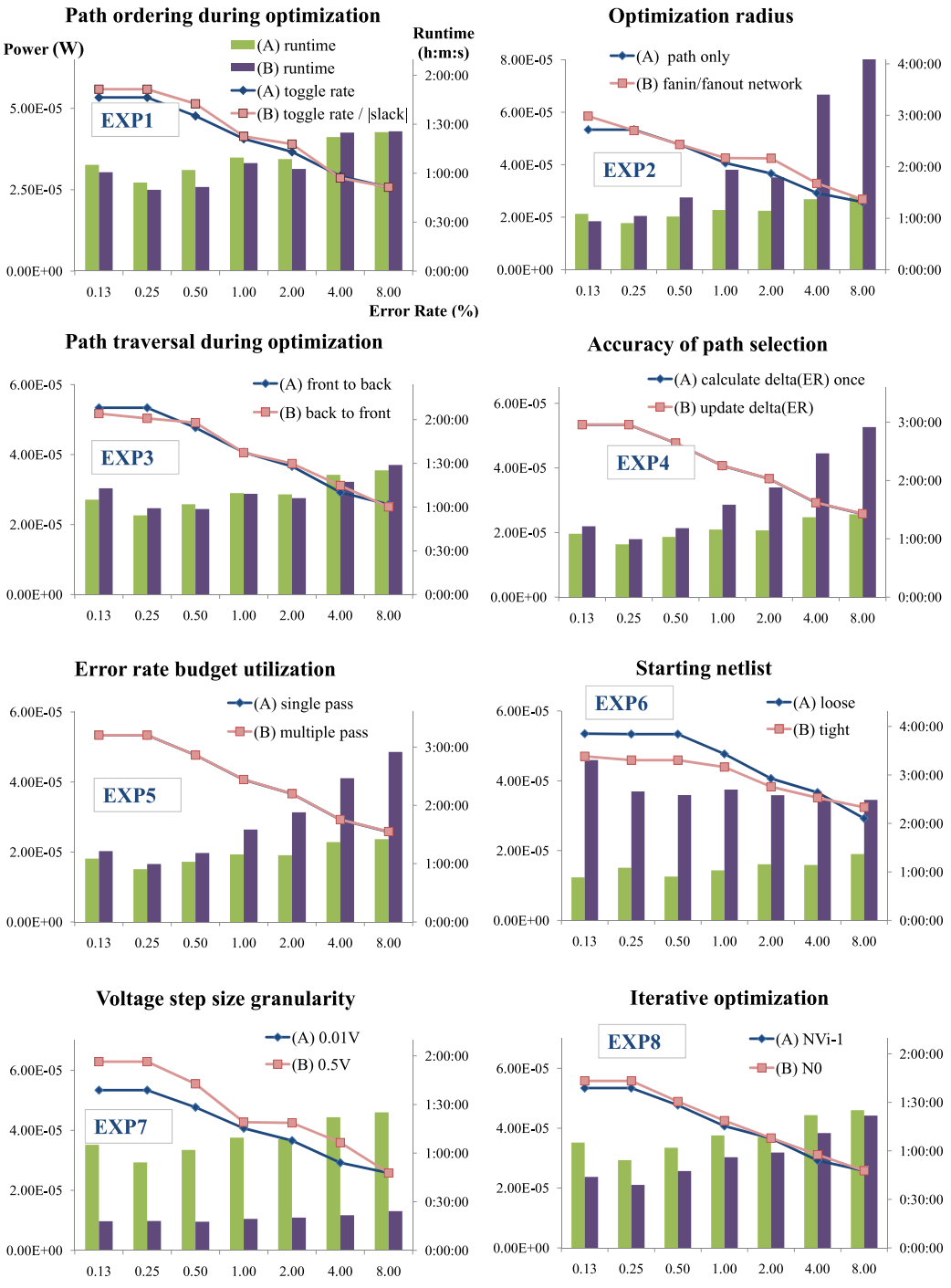


Figure 4.10: Evaluation of different heuristic design choices. The choices are evaluated in terms of power of the resulting design as well as runtime.

procedures – *OptimizePaths* (cell upsizing to reduce the error rate) and *ReducePower* (cell downsizing to reduce area and power). When starting the optimization flow from a loosely constrained design, path optimization provides the most substantial contribution to power reduction by reducing the error rate and extending voltage scaling. However, when starting from a tightly constrained design, much optimization has already been performed, and the power reduction stage of our heuristic is essential for power minimization. Although runtime increases due to evaluation of more downsizing moves, a tightly constrained netlist provides a better starting point, since it permits more voltage scaling. Voltage scaling has a stronger effect on power reduction and scales the power of all cells, while area reduction only affects the downsized cells. Also, starting from a tightly constrained design reduces the dependence on ECO, which improves the optimization efficiency. Using a coarser-granularity voltage step reduces runtime significantly, but comes at the cost of power, since the heuristic cannot home in on the optimal voltage as easily. For higher error rates, a large step size can provide a near-optimal power result and a large reduction in runtime. Thus, error rate-aware adaptive step sizing can be beneficial.

In terms of **iterative optimization**, we observe that our heuristic is able to achieve the same result independent of the starting netlist. Thus, we choose the option that minimizes runtime.

4.5.2 Comparison Against Alternative Flows

To demonstrate the benefits of our recovery-driven design flow, we compare five alternative design flows – traditional placement and routing (P&R) implementations with conventional and tight timing constraints, a BlueShift-like path constraint tuning (PCT) approach, gradual slack design [33] [32], and our heuristic for error rate-optimized recovery-driven design. Figure 4.11 compares the power consumptions of the various design techniques at several target error rates.

Recovery-driven designs reduce power by enabling extended voltage scaling and keeping area overhead low with respect to other optimization techniques. Compared to a conventionally optimized design, a recovery-driven design operates at a much lower voltage for a given target error rate, due

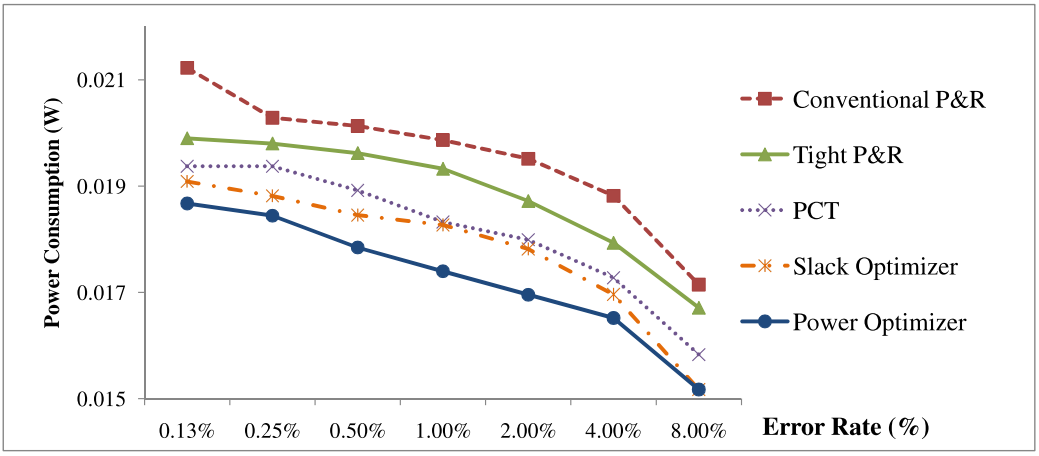


Figure 4.11: Power consumption of each design technique at various target error rates for target modules in Table 4.1.

to the functionally-aware optimization approach that optimizes the paths that cause the most errors. Compared against a highly optimized design that uses tightly constrained P&R, a recovery-driven design reduces power by minimizing the amount of area spent on path optimization. Traditional tightly constrained designs are functionally agnostic and optimize all paths heavily, incurring a large area overhead. Recovery-driven designs, on the other hand, use functional information to target only the paths that cause the most errors, thereby minimizing the area cost of additional voltage scaling. In scenarios where the cost of area is high, such as for technologies with higher leakage like those forecast in future technology generations, the cost of functionally-agnostic optimizations will increase, and the benefits of recovery-driven design will increase. Table 4.4 shows power savings for recovery-driven design for each module with respect to traditional P&R at different target error rates.

In our power minimization heuristic, after deciding how to allocate the error rate budget, the *ReducePower* stage performs aggressive cell downsizing to reduce circuit area and power. Table 4.5 compares recovery-driven design against other design flows in terms of area overhead with respect to the baseline design. Design for a target error rate has similar area overhead to PCT but still produces a design with lower power. The reason is that designing for a target error rate allows more aggressive voltage scaling before the target error rate is exceeded. At lower voltages, there are more negative slack paths to be optimized during *OptimizePaths*, which increases area overhead.

Table 4.4: Power savings (%) for error rate-optimized recovery-driven designs compared to traditional P&R.

MODULE	Target Error Rate (ER_{target})						
	0.125%	0.25%	0.5%	1.0%	2.0%	4.0%	8.0%
lsu_dctl	29.1	16.8	16.8	16.8	16.8	16.8	21.6
lsu_qctl1	8.8	6.7	5.8	8.1	11.0	9.0	8.6
lsu_stb_ctl	17.9	17.9	18.1	15.4	9.6	19.2	2.9
sparc_exu_ecl	6.0	6.0	18.3	18.3	22.7	23.3	17.4
sparc_ifu_dec	13.7	10.1	8.6	14.3	15.9	18.5	15.1
sparc_ifu_errdp	2.2	2.8	5.7	5.7	5.7	9.3	9.3
sparc_ifu_fcl	14.5	15.4	16.5	19.2	19.2	19.2	19.2
spu_ctl	13.1	13.1	13.1	13.2	8.8	1.6	8.9
tlu_mmu_ctl	0.8	0.8	0.8	0.8	0.8	0.8	0.8

Table 4.5: Average area overhead with respect to the baseline.

Tight P&R	PCT	SlackOpt	PwrOpt 0.125%	PwrOpt 0.25%
19.1%	5.0%	11.9%	3.9%	4.3%
PwrOpt 0.5%	PwrOpt 1%	PwrOpt 2%	PwrOpt 4%	PwrOpt 8%
4.8%	5.4%	5.8%	6.0%	5.3%

However, aggressive downsizing keeps area overhead low; and since the paths targeted by *PowerOptimizer* cause the most errors in the design, the area is well spent, and the additional voltage scaling contributes to a net benefit in terms of power savings. PCT, on the other hand, adds tighter timing constraints to the registers where the most errors are captured and optimizes all paths with endpoints at those registers. Since our heuristic targets paths individually, we can target the error-causing paths more efficiently, reduce overhead, and increase voltage scaling and power savings.

Compared to tightly constrained P&R and gradual slack design, design for a target error rate incurs significantly less area overhead and reduces power. On one hand, tightly constrained P&R is functionally agnostic and fails to identify the set of paths that maximizes voltage overscaling per unit area overhead. Gradual slack design, on the other hand, optimizes the design to make tradeoffs between power, throughput, and reliability over a *range* of error rates. Thus, a gradual slack design is over-optimized for any single

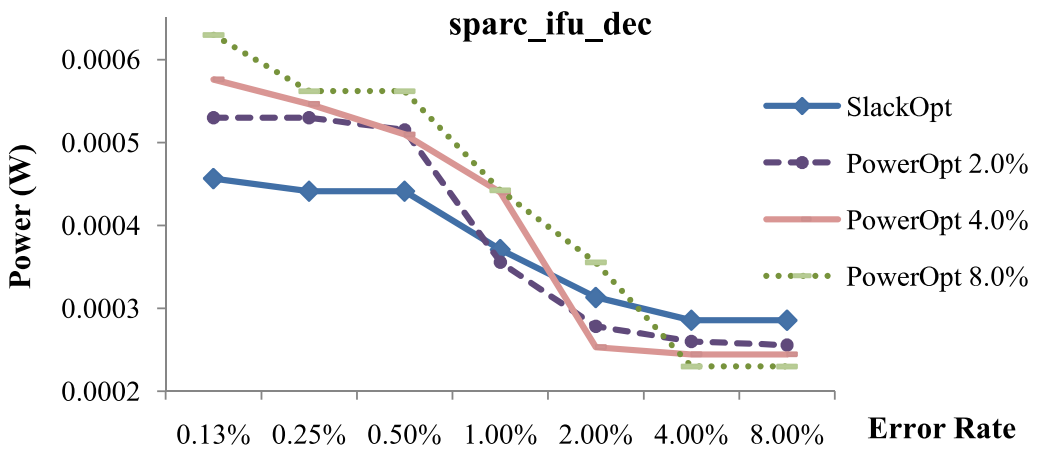


Figure 4.12: Recovery-driven design for a target error rate (*PowerOpt*) minimizes power at the target error rate. Gradual slack design (*SlackOpt*) optimizes a design for a range of error rates to provide adaptability and smooth performance-power tradeoffs.

target error rate.

Figure 4.12 compares recovery-driven design for a target error rate against gradual slack design. The results show that designing for a target error rate minimizes power at the target error rate. However, since a recovery-driven design can have a non-zero error rate even under nominal conditions, power efficiency at error rates lower than the target may drop off steeply. Likewise, since design for a target error rate creates a slack wall at the error-optimal voltage, additional benefits for error rates higher than the target are limited. A gradual slack design, on the other hand, is optimized for a *range* of error rates. Although this means that it is less efficient than an error rate-optimal design for any single error rate, it also means that performance or output quality can be efficiently traded for power savings over the entire range of error rates. Thus, whenever more errors can be tolerated, a gradual slack design can reduce power consumption. This may not be possible for an error rate-optimal design, since it forgoes scalability to achieve additional power savings at the target error rate.

Recovery-driven design optimizes for errors in the average operating behavior of a design. If the frequently exercised paths during operation are significantly different than those targeted during optimization, then too many errors may be produced, and voltage scaling may be limited for a target error rate. To evaluate the robustness of recovery-driven design when the

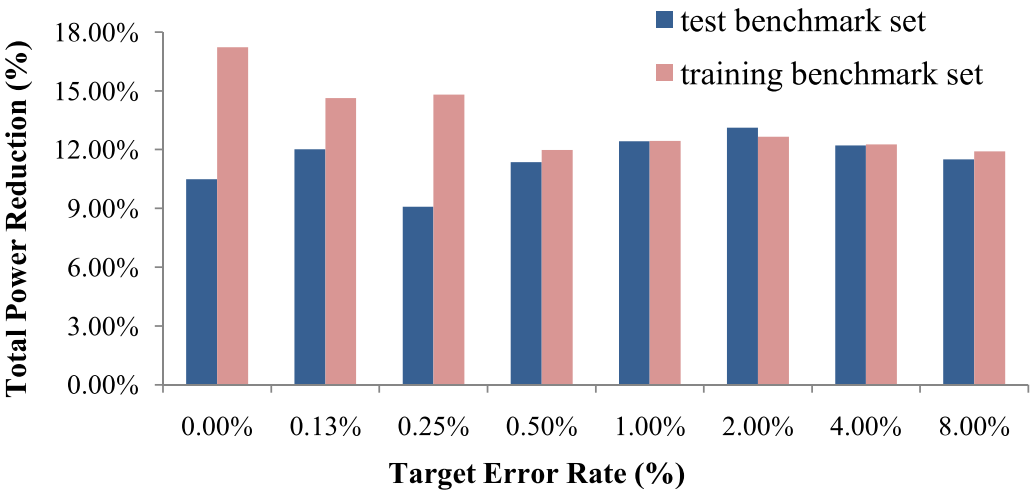


Figure 4.13: Total power reduction over tightly constrained design for the training (optimization) and test benchmark sets. Power reductions for the training set are slightly higher, since the design has been optimized specifically for the activity profile of this set.

workload changes, we compared the power reduction achieved when running the training (optimization) benchmarks against power reduction for the test benchmarks. Figure 4.13 shows that power reduction is slightly higher for the benchmark set that the processor was optimized for, but the difference is only about 1% on average.

4.5.3 Variation-aware Analysis

Recovery-driven design increases energy efficiency by reshaping the slack distribution of a design, such that error rate is reduced at a particular voltage. Figure 4.14 shows activity-weighted slack distributions (sum of path toggle rate vs. timing slack) from before and after optimization, confirming that the optimization increases slack for frequently exercised paths, which enables extended voltage scaling for a target error rate. However, due to random variations introduced in the physical circuit by sources of static and dynamic non-determinism, the actual slack distribution may be somewhat different than the designed distribution.

To test the benefits of recovery-driven design in the presence of variations, we have implemented a model for inter-die and spatially correlated within-die variations based on the models in [60, 61]. We use an exponential model for

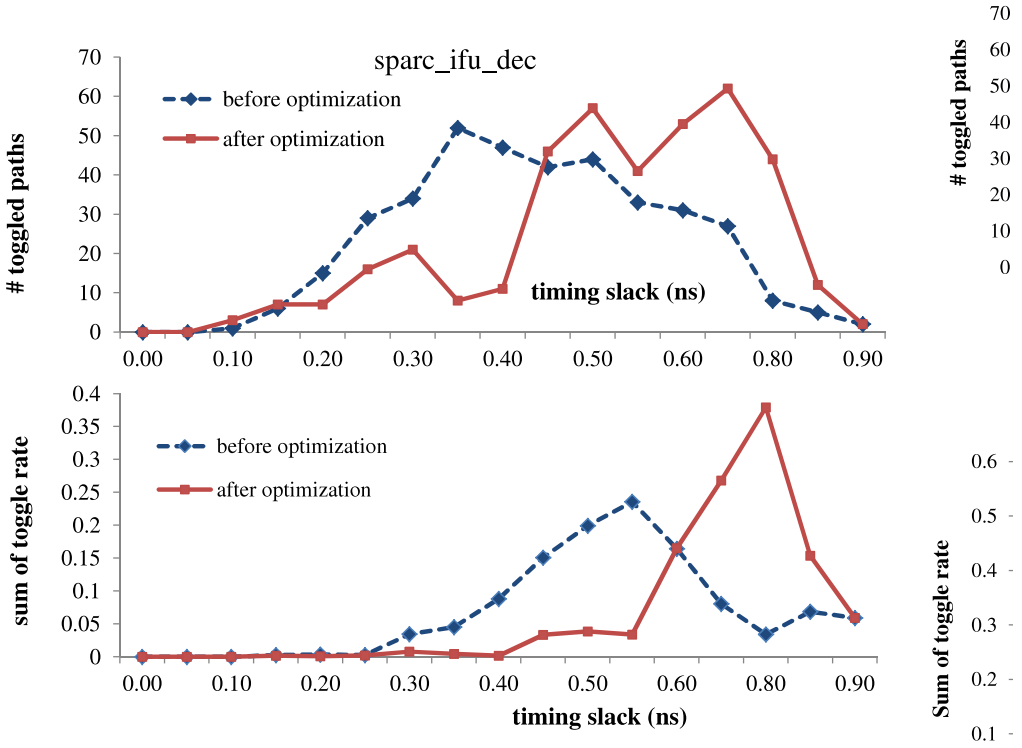


Figure 4.14: Recovery-driven design reshapes the slack distribution by adding slack to frequently exercised paths and removing slack from infrequently exercised paths. The activity-weighted slack distribution (bottom) shows the sum of toggle rates for all paths with a particular slack value, confirming that frequently exercised paths have more slack in the optimized netlist.

correlation between different die locations, in which the correlation function decays exponentially as a function of distance, with parameters supplied by the authors of [60]. We extract standard deviations (σ) of cell delay at each operating voltage from SPICE simulations, and use our variation model to assign a random delay variation to each die and each gate within the die, based on its location. We then repeat error rate and power estimation with one hundred different random variation maps. From the Monte Carlo simulations, we report total power consumption of the target modules at each error rate in Table 4.6. Table 4.6 shows that even when variations are accounted for, recovery-driven design still achieves significant power savings over a conventional design. Furthermore, the average benefits do not noticeably change when variations are accounted for. (Power reduction in Table 4.6 is somewhat lower for error rates below 1% because the test design was optimized for a target error rate of 1%.) Random variations cause perturbations within a design but do not shift the average case behavior. Since recovery-driven designs are optimized for and operate at the average case operating point, they are naturally robust to random variations.

Since conventional designs do not afford special optimization to frequently exercised paths, they are more likely to be critical paths in a conventional design than in a recovery-driven design. Thus, the operating voltage for a non-zero error rate is primarily determined by accrual of errors on the frequently exercised paths. Also, since the frequently exercised paths are more likely to be timing critical in a conventionally optimized design, there is a higher chance that variations impact the operating voltage of the processor by impacting the delay of frequently exercised paths. Note that even if variations reduce delay on some frequently exercised paths, relatively few frequently exercised paths with increased delay can force the design to a higher operating voltage.

Recovery-driven design, on the other hand, affords additional slack to frequently exercised paths, so that the operating voltage for a target error rate is primarily determined by a slow accrual of errors on the infrequently exercised paths. Since there are typically many more infrequently exercised paths than frequently exercised paths, random variations mostly have an averaging effect, such that recovery-driven designs are fairly robust to variations. Thus, variation analysis reveals that recovery-driven designs are actually more robust to variations than conventionally optimized designs, since it is common

Table 4.6: Variation-aware analysis.

	Target error rate (ER_{target})						
	0.125%	0.25%	0.5%	1.0%	2.0%	4.0%	8.0%
Power consumption (W) in baseline design							
Min.	0.0126	0.0126	0.0122	0.0113	0.0108	0.0106	0.0095
Max.	0.0202	0.0201	0.0199	0.0196	0.0191	0.0186	0.0167
Avg.	0.0162	0.0160	0.0156	0.0153	0.0149	0.0141	0.0127
Power consumption (W) in recovery-driven design							
Min.	0.0111	0.0106	0.0105	0.0096	0.0092	0.0088	0.0080
Max.	0.0187	0.0183	0.0175	0.0172	0.0165	0.0161	0.0151
Avg.	0.0148	0.0144	0.0141	0.0134	0.0128	0.0123	0.0113
Power reduction (%)							
Avg.	8.28	9.71	9.43	12.61	13.80	13.03	11.18

for variations to increase the power (operating voltage) of a conventional design, but uncommon for variations to increase the power (operating voltage) of a recovery-driven design.

4.5.4 Recovery-driven Processors

In this section, we demonstrate the benefit of designing processors for specific hardware and software error resilience mechanisms, as described in Section 4.3.

Circuit-Level Timing Speculation. Figure 4.15 compares the energy consumption of a recovery-driven processor that has been designed and optimized for Razor against the power consumption of processors designed for other objectives, such as gradual slack or PCT, and against processors that have been designed for correctness but use the traditional Razor methodology to save energy. We assume a recovery overhead of nine cycles, proportional to the pipeline depth of the processor.

Figure 4.15 demonstrates that the minimum energy is indeed achieved by a processor that is designed to produce errors that can be gainfully tolerated by Razor. Designing the processor for the error rate target at which Razor operates most efficiently allowed us to extend the range of voltage scaling from 0.84 V for the best "designed for correct operation" processor to 0.71 V

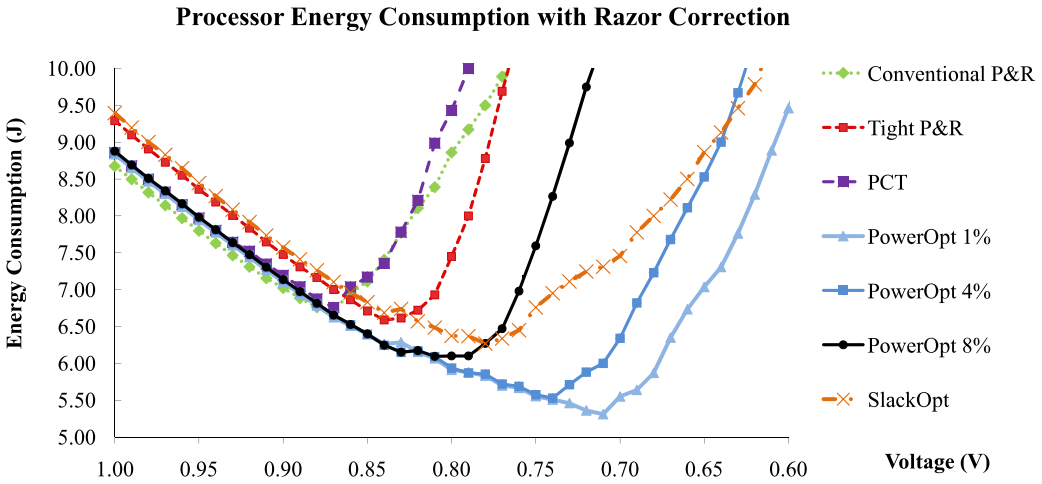


Figure 4.15: The benefit of designing a processor to produce errors then correcting them with an error tolerance mechanism over designing for correctness and then relaxing the correctness guarantee can be significant. Results are shown for processors that employ Razor.

for the processor designed for an error rate of 1%, affording an additional 19% energy reduction.

Error recovery with a circuit-level approach like Razor imposes a throughput penalty, since error recovery requires feeding correct values back into the pipeline. Figure 4.16 shows the throughput reduction caused by error recovery for a correction overhead of 5 cycles. As can be seen, a recovery-driven processor even minimizes the recovery overhead at the target operating voltage.

Application Noise Tolerance. To demonstrate the benefits of recovery-driven design targeted at application-level noise tolerance, we use a face detection algorithm [44] as the example application. Face detection is naturally robust to errors in several processor modules and does not require strict computational correctness. Rather than causing program failure, errors may result in reduced output quality (false positive or negative detections) [62].

Face detection, as well as the other error-tolerant applications we consider, tolerates errors in the arithmetic units of the processor. For this class of applications (which relies heavily on numerical computation), the arithmetic units account for approximately 35% of the dynamic power consumption of the processor.

Figures 4.17 and 4.18 compare the power consumption of processors de-

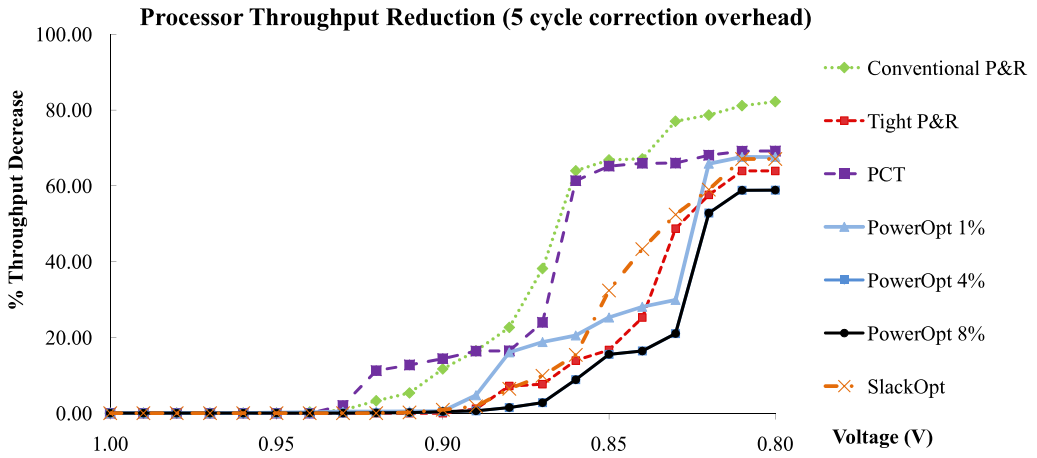


Figure 4.16: Throughput reduction at different voltages for an error recovery overhead of 5 cycles. This recovery overhead is appropriate for a simple pipeline or lightweight recovery technique.

signed for application-level error tolerance of arithmetic errors using single- and dual-voltage rail designs, as described in Section 4.3. In these figures, all processors achieve the same output quality at a given error rate, but processors designed to allow errors consume less power, and power is minimized for these designs at their respective error rate targets. For example, at an error rate of 1%, where output quality is still maximized for the face detection application, the processor designed for an error rate target of 1% consumes 19% less power for dual-rail design and 15% less power for single-rail design than the baseline correctness-optimized processor. Benefits are even higher for larger error rates if some application output degradation is permissible.

Note that we can always perform error-free computation on a core designed for application-level noise tolerance by scaling down the frequency to the point where all paths have non-negative slack. However, this may represent a performance penalty when compared to relaxed-correctness operation.

Also note that trends in processor-level results may differ somewhat from trends in averaged module-level results. Whereas the power reduction of a recovery-driven design is limited by a module's critical paths, the power reduction of a recovery-driven processor is biased by the critical modules that begin causing errors first when voltage is scaled down. As we will show in the next section, results can be improved by utilizing multiple voltage domains.

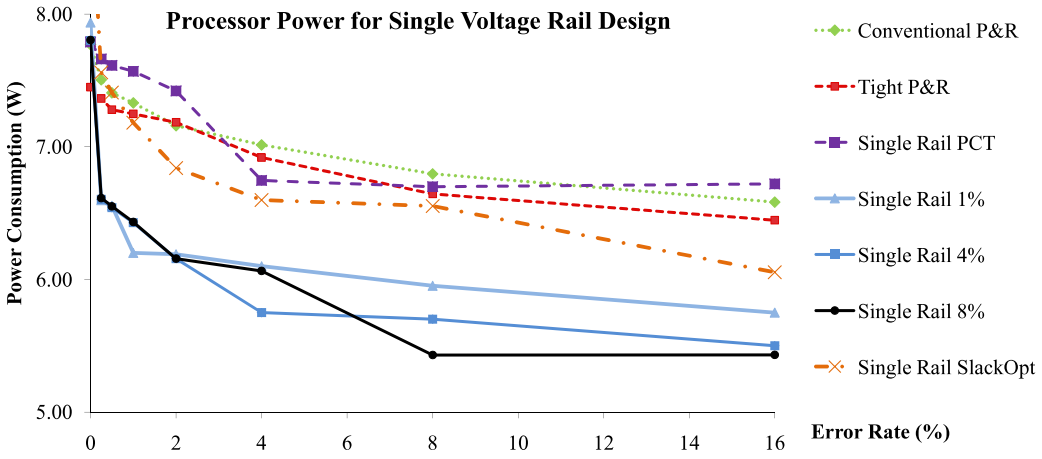


Figure 4.17: This figure demonstrates the power benefit of a processor that is designed to allow errors in the arithmetic units over a processor that is designed for correctness. All modules in the processor operate at the same voltage. Razor is used to correct errors in non-arithmetic units.

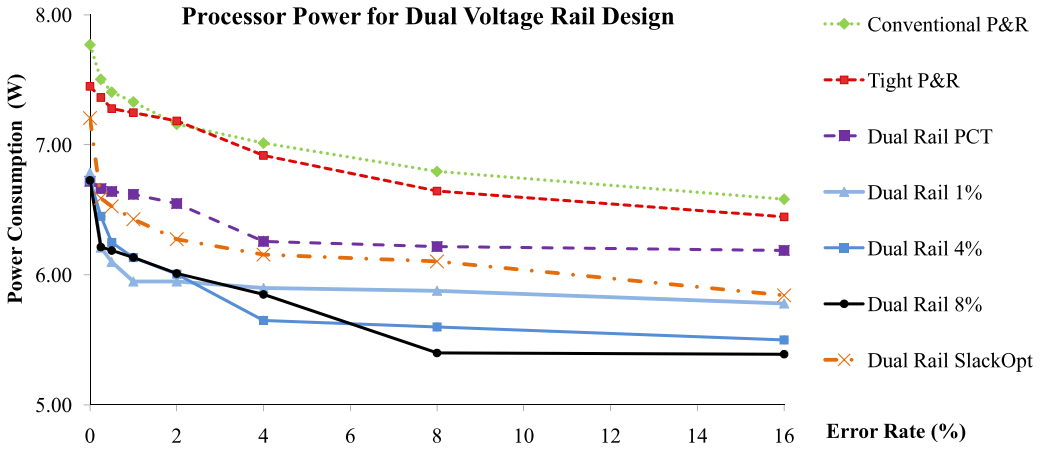


Figure 4.18: This figure demonstrates the power benefit of a processor that is designed to allow errors in the arithmetic units over a processor that is designed for correctness. The processor uses a dual voltage rail design with the arithmetic units on a separate rail.

4.5.5 Supporting Multiple Voltage Domains

Given a target error rate, the module-level power minimization heuristic in [48] selects an optimal operating voltage for a processor module. However, the proposed processor core-level methodology (Algorithm 1, $DOMAINS = 1$) selects a common voltage for all modules of a processor core. Table 4.7 shows that different modules vary (sometimes substantially) in their optimal voltage operating points due to a number of factors, including module area (number of paths and cells), slack distribution (fraction of paths that are critical), and activity factor (how often paths toggle). In addition, the table shows that the range of optimal module voltages increases when designing for a non-zero error rate target.

Because of the above module-level variations, there can be a substantial difference in terms of power consumption between the locally and globally optimized module implementations. Figure 4.19 quantifies the difference between single and multiple voltage domain designs for processor cores tolerating different error rates. We compare designs with different numbers of voltage domains, targeting different processor error rates in terms of their power consumption relative to a processor optimized for a common operating voltage. The results show that the power efficiency of recovery-driven processors will improve significantly with the number of voltage domains that are supported. In practice, the number of voltage domains should be chosen by carefully balancing the voltage overscaling benefits with the area and complexity overheads of supporting multiple power rails. The results of Figure 4.19 do not consider the overhead of level shifter circuitry.

4.5.6 Robustness to Application Diversity

Different workloads exercise the timing paths of a processor core differently. Thus, the sets of frequently exercised and infrequently exercised paths may change, depending on the workload. Since recovery-driven designs are optimized according to an average case activity profile, it is important to ensure that power efficiency is not degraded significantly when the activity profile of a workload is not the same as the activity profile for which the processor was optimized.

To gauge the robustness of recovery-driven design to workload diversity,

Table 4.7: Optimal module voltages at different target error rates.

MODULE	Target error rate (ER_{target})						
	0.0%	0.125%	0.25%	0.5%	1.0%	2.0%	4.0%
lsu_dctl	0.75	0.72	0.71	0.75	0.74	0.73	0.72
lsu_qctl1	0.88	0.87	0.86	0.85	0.84	0.83	0.80
lsu_stb_ctl	0.77	0.76	0.75	0.75	0.70	0.68	0.66
sparc_exu_ecl	0.75	0.74	0.73	0.70	0.70	0.69	0.70
sparc_ifu_dec	0.68	0.67	0.66	0.63	0.70	0.58	0.57
sparc_ifu_errdp	0.77	0.58	0.57	0.56	0.55	0.54	0.53
sparc_ifu_fcl	0.79	0.77	0.76	0.75	0.74	0.73	0.72
spu_ctl	0.78	0.65	0.64	0.63	0.62	0.63	0.58
tlu_mmu_ctl	0.85	0.52	0.51	0.51	0.51	0.51	0.51
RANGE	0.20	0.35	0.35	0.34	0.33	0.32	0.29

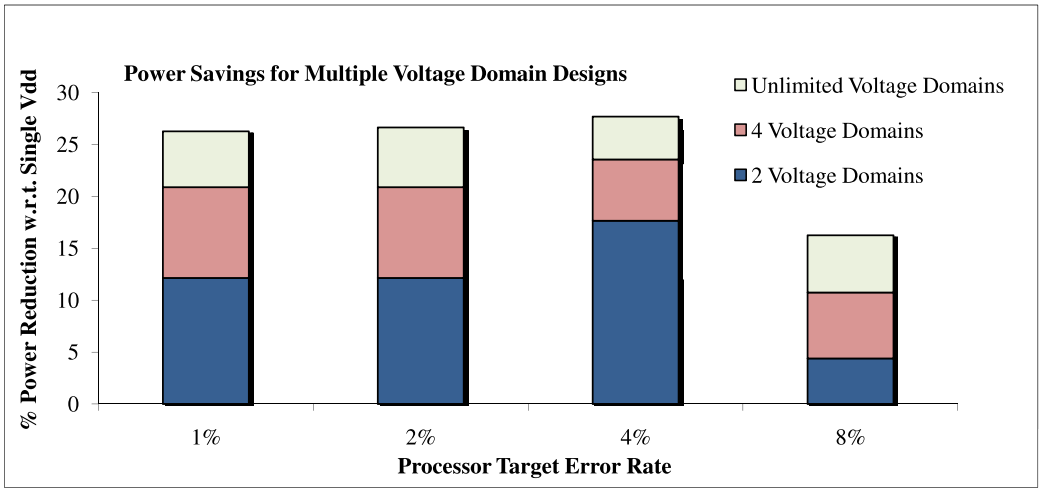


Figure 4.19: The benefit of a multiple voltage domain design over a single voltage domain design can be significant when designing for an error rate target. Substantial power savings can be achieved when each module is optimized for a locally optimal voltage rather than the globally optimal voltage of the module group. The stacked bars show the additional power savings afforded as the number of voltage domains increases.

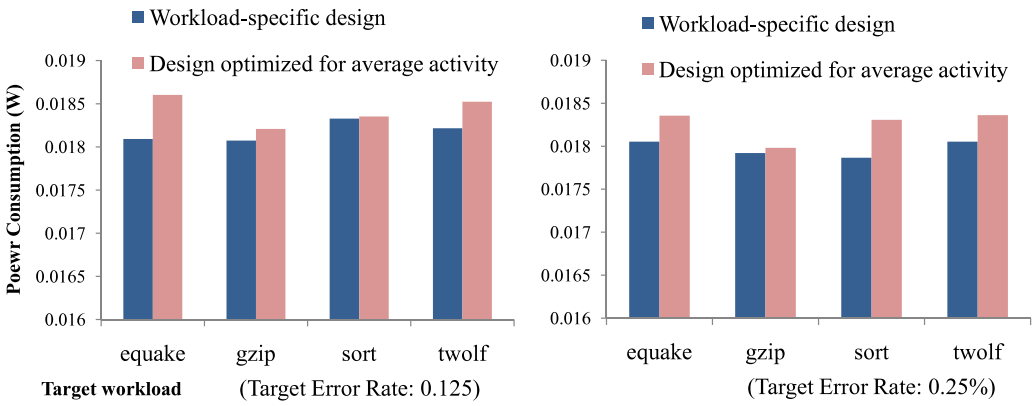


Figure 4.20: Recovery-driven design is robust to application diversity. On average, processor modules that have been optimized for the average case only consume 1% more power than modules that have been customized specifically for the activity profile of the test workload.

we create several recovery-driven designs, optimized for the activity profiles of each benchmark in the test set – *equake*, *gzip*, *sort*, and *twolf*. Then, we compare the power consumption of each benchmark in the test set, running on the design that was optimized for the average case, against the design that was optimized specifically for that benchmark. Figure 4.20 compares the power consumption of average case design against workload-specific designs for different target error rates.

On average, the difference is small – only 1.5% difference in power at an error rate of 0.125% and 0.9% difference at 0.25% – demonstrating the robustness of recovery-driven design to application diversity. The difference will decrease as the target error rate increases. The reason for this robustness is that since some paths are allowed to cause errors, there is some "forgiveness" when the priority of path optimization deviates somewhat from the optimal. Our recovery-driven design heuristic bins paths into P_- paths that are allowed to cause errors and P_+ paths that should remain error-free. As long as the difference in activity for a path is not so much as to make the path switch bins, the path dichotomy is preserved and power efficiency is not degraded. In the worst case, we only observe 3% degradation in power efficiency.

4.6 Related Work

Design-level Optimization.

The design-level optimizations proposed in this chapter target the most frequently exercised paths in a stochastic design for special optimizations, since highly exercised paths have the potential to cause the most errors. EVAL [63] is a design-level optimization that aims to increase the amount that frequency can be overscaled in a timing speculative design. EVAL attempts to increase the efficiency of frequency overscaling by optimizing the most frequently exercised paths in a design at the expense of the majority of the static paths. Consequently, errors are prevented on the highly active paths (that would cause many errors) and allowed on the infrequently exercised paths (that cause relatively few errors).

EVAL trades error rate for frequency by shifting, tilting, or reshaping the path slack distributions of the various functional units in a design.

BlueShift [64] is an application of EVAL that also optimizes a circuit for frequency overscaled operation. BlueShift uses EVAL techniques in an iterative optimization in an attempt to reduce the error rate of a circuit module. In each iteration, some optimizations are performed, and the error rate is checked. If the error rate is less than the target rate, the module optimization finishes. Otherwise, optimization iterations continue.

Each optimization step involves adding timing slack to the paths that cause the most timing errors and performing a gate-level simulation to check whether the path adjustments have brought the error rate below the target. BlueShift uses two methods to add slack to frequently exercised circuit paths: (1) forward body biasing of selected gates and (2) path constraint tuning that applies tighter timing constraints for selected paths.

Our work differs from BlueShift in objective, approach, and scope of optimization. Our objective is to minimize power, while BlueShift's objective is to improve performance. Consequently, we use sensitivity functions that are voltage-aware. Also, BlueShift requires iterative gate-level simulation and re-layout, making the approach time-consuming and impractical for large modern SOC designs, where the number of post-sizing layout, extraction, and simulation steps is often limited by runtime constraints. Furthermore, while BlueShift optimizes only the post-synthesis circuit over many layout iterations, our recovery-driven design techniques perform both activity-guided

post-synthesis and post-layout optimizations in a single pass to enhance energy efficiency.

Dynatune [65] is another stochastic optimization technique, similar to previously mentioned techniques, that improves performance by enhancing the efficiency of frequency overscaling-based timing speculation. Rather than treating all gates in a design as equals, Dynatune focuses optimization efforts on the most dynamically critical cells. The most dynamically critical cells are those with the highest switching activity. Dynatune starts with a circuit design that is implemented with high V_t cells (which have high delay and low leakage), and replaces some of the most dynamically critical cells with low V_t cells, thereby reducing their delay but increasing their leakage power consumption. Dynatune replaces as many cells as possible while staying below a specified threshold for leakage power. Since some paths that remain with many high V_t cells may not meet timing after optimization is finished, the design is timing speculative and must incorporate an error recovery mechanism to cover the case when timing speculation results in errors. Therefore, the goal of Dynatune is to assign low V_t cells, based on dynamic criticality, to maximize the peak frequency of a timing speculative design while staying within a given leakage power budget.

Like other stochastic optimizations discussed in this section, work on better-than-worst-case (BTWC) logic synthesis [66] has also proposed to use activity information to reduce the error rate of an overscaled design. Whereas traditional synthesis tools attempt to minimize delay for a logic block, a BTWC synthesis tool also considers switching probabilities when implementing a design. Reducing switching activity can result in fewer errors for an overscaled design.

BTWC logic synthesis uses functional information to reduce the probability of error for scenarios in which multiple possible logic decompositions for a block have equal cost. In such cases, ties between the original cost function (delay) are broken by a new cost function that takes switching probability into account. The tiebreaker cost function is a sum of delay, weighted by switching probability. Thus, the logically equivalent decomposition with the least switching activity (i.e., the most biased signal probabilities) is selected in the event of a tie.

Some limitations of BTWC synthesis in relation to other stochastic optimizations are the inability to target a specific error rate and potentially

limited impact, since only logic with multiple candidate implementations that have the same delay and functionality are optimized.

Sensitivity-Based Cell Sizing. Our methodology relies on cell sizing for slack distribution. Sensitivity-based downsizing approaches have been proposed in previous literature[45, 46, 67, 68, 69, 70]. TILOS [67] proposes a heuristic that sizes transistors iteratively, according to the sensitivity of the critical path delay to the transistor sizes, in order to find an optimum (with maximum delay reduction per transistor width increase). Equation (4.4) shows the sensitivity function of TILOS. ΔL and ΔD represent the change in leakage and delay for a resized transistor. The techniques proposed in [69] use the same sensitivity function as $TILOS$.

$$Sensitivity = \Delta L / \Delta D \tag{4.4}$$

For the cell sizing in [70], all cells are sorted in decreasing order of $\Delta L \times S$, where ΔL is the improvement in leakage after a cell is replaced with its less leaky variant, and S is its timing slack after the replacement has been made. The techniques proposed in [45, 46] use sensitivity-based *downsizing* (i.e., begin with all nominal cell variants and replace cells on non-critical paths with long channel-length variants) heuristics for leakage optimization. The heuristics defined the sensitivity associated with a cell instance as follows.

$$Sensitivity = \Delta L / \Delta S \tag{4.5}$$

In Equation (4.5), ΔS represents the slack change of a given cell instance after downsizing. ΔL indicates the leakage change of cell instance after downsizing. The sensitivities are computed for all cell instances. The heuristics of [45, 46] select a cell with the largest sensitivity and perform downsizing with a logically equivalent cell. If there is no timing violation in incremental STA, this move is accepted and saved.

Our work uses cell sizing in a novel context – as a mechanism to optimize hardware for non-zero error rates.

4.7 Summary

In this chapter, we propose *recovery-driven design*, a design-level approach that optimizes a processor module for a target timing error rate instead of correct operation. We present a detailed evaluation and analysis of a recovery-driven design methodology that minimizes power for a target error rate. We extend our recovery-driven design flow to design recovery-driven processors – processors that are designed and optimized for a target error rate. We also present an extension of our recovery-driven design flow that creates a gradual slack design that is optimized for a range of error rates rather than a single target. The gradual slack technique is used to design soft processors that can trade throughput or output quality for energy savings over a range of reliability targets. While we have chosen to focus on improving the energy efficiency of error-resilient designs, recovery-driven design can also be used to optimize other design characteristics, such as yield.

CHAPTER 5

ARCHITECTURE-LEVEL OPTIMIZATION OF PROGRAMMABLE STOCHASTIC PROCESSORS

Previous chapters have described circuit or design-level optimizations that manipulate the error rate behavior of a design to increase the energy efficiency of operating at a nen-zero error rate. In this chapter, we investigate whether architectural optimizations can also manipulate error rate behavior to significantly improve the energy efficiency of operating at a non-zero error rate. To this end, we demonstrate how error rate behavior indeed depends on processor architecture, and that architectural optimizations can be used to manipulate the error rate behavior of a processor. Using architecture-level optimizations for programmable stochastic processors, we demonstrate enhanced overscaling and up to 29% additional energy savings for processors that employ Razor-based timing speculation.

5.1 Introduction

Traditionally, processors have been architected to operate correctly under worst-case operating conditions. Ensuring timing correctness under all possible circumstances requires that conservative guardbands be imposed on operating frequency and voltage, limiting the performance and energy efficiency of modern processor designs, especially as device feature sizes continue to shrink and the impact of process and dynamic variations escalates. The growing costs of providing the illusion of perfect hardware on top of increasingly stochastic and unreliable devices have become prohibitive. To counter the rising costs of variability more efficiently, several timing speculative error-resilient design techniques have been proposed [19, 20, 21, 71, 22]. These techniques relax correctness guards to gain efficiency in the average case at the expense of some errors. Errors are corrected or tolerated by hardware or software error resilience mechanisms to maintain the level of output

quality expected by the user.

The magnitude of efficiency benefits available from timing speculation depends on two factors – where and how often the processor produces errors when operating at an overscaled voltage or frequency. If the frequency of errors can be reduced for a timing speculative design, the range of overscaling can be extended, affording additional energy or performance gains. Previous works have demonstrated the potential to increase the energy efficiency [32, 33, 48] or performance [63, 64] benefits of timing speculation by modifying the error distribution of a timing error-resilient design. However, these works focused only on circuit-level techniques. It remains to be shown whether architecture-level optimizations can similarly affect the error distribution of a timing speculative design to generate energy or performance gains.

In this work, we demonstrate that the error distribution indeed depends on architecture. We show that the error distribution of a design that has been architected for error-free operation may limit scalability and energy efficiency for better-than-worst-case operation. Thus, optimizing architecture for correctness can result in significant inefficiency when the actual intent is to perform timing speculation. In other words, one would make different, sometimes counterintuitive, architectural design choices to optimize the error distribution of a processor to exploit timing error resilience. Thus, we make a case for timing error resilience-aware architectures and propose architectural optimizations that improve the efficiency of timing speculation.

This work on timing error resilience-aware architecture makes the following contributions.

- We show that the error distribution of a timing speculative processor strongly depends on its architecture. As such, we demonstrate that architectural optimizations can be used to significantly improve the efficiency of timing speculation.

- We confirm, with experimental results for different implementations of a 4-tap FIR filter, as well as Alpha, MIPS [72], FabScalar [73], and OpenSPARC [52] processor cores, that timing error resilience-aware architectural design decisions can indeed significantly increase the efficiency of a timing speculative architecture.

Note that we have used voltage overscaling as the proxy for all variation-induced errors in this chapter. Our analysis and conclusions should apply for other sources of timing variation as well.

The rest of the chapter is organized as follows. Section 5.2 describes the architectures that we evaluate and provides examples of how architectural decisions can influence the slack and activity distributions of a design. Section 5.3 describes our experimental methodology. Section 5.4 presents results and analysis showing that optimizing an architecture for timing speculation can significantly improve energy efficiency. Section 5.5 discusses related work. Section 5.6 summarizes the chapter.

5.2 Understanding and Manipulating the Error Distribution of Timing Speculative Architectures

As explained in Section 4.1.1, the timing error rate of a processor depends on its slack and activity distributions. In this section, we argue that both the slack and activity distributions of processors are strongly dependent on processor architecture. This dependence implies that architectural features can be chosen to optimize the slack and activity distributions and, by extension, the error distribution and energy efficiency of a timing speculative processor. First, we demonstrate how slack and activity distributions depend on processor architecture. Then, we show how architectural optimizations can change the slack and activity distributions.

5.2.1 Architectural Dependence of Slack and Activity Distributions

In this section, we show that slack and activity distributions indeed depend on architecture. First, we present four functionally equivalent architectural variants of a 4-tap FIR filter We describe how the architectural characteristics of each filter determine the properties of its slack and activity distributions.

The baseline FIR filter, shown in Figure 5.1(a), is the simplest and most well-known arrangement of the FIR filter architecture, containing four MAC units. A pipelined version of the filter (Figure 5.1(b)) was created by creating a cutset across the outputs of the multipliers and adding a latch to each arc.

We also created a folded version of the filter (Figure 5.1(c)), in which multiple operations are mapped to a single hardware unit. Folding by a factor of two multiplexes the filter hardware so that half of the filter coefficients are active in even cycles, the other half are active in odd cycles, and an output sample is computed every two cycles. The blocked architecture of Figure 5.1(d) was created by replicating the internal filter structure to compute two samples in parallel.

Figure 5.2 compares the path slack distributions of the different filter implementations, confirming our intuition that the slack distributions of the filter designs depend strongly on the architecture. Table 5.1 presents more detailed information on how slack and activity change for different architectures. The mean and standard deviation of the slack distribution (μ_{slack} and σ_{slack}, respectively) tell how much initial slack exists, on average, and how regular the slack distribution is, i.e., how spread out the values of path delay are. Designs with more regular (less spread) slack distributions allow less overscaling past the critical point because a large number of paths fail at the same time, potentially causing a steep increase in error rate. The average path activity (α_{path}) shows how frequently paths toggle. Higher path activity can mean that error rate increases more steeply, since negative slack paths generate more errors when they toggle more frequently.

Table 5.1 and Figure 5.2 reveal that pipelined and folded architectures have more regular slack distributions. These architectures have shorter paths that have less capacitance, less delay sensitivity to voltage scaling, and less variation in absolute path delay. This creates extra slack compared to other architectures, but limits scaling past the critical point. The folded architecture has high path activity, since the internal filter elements must operate at twice the frequency of the baseline design to achieve the same sample rate. Likewise, the blocked architecture has reduced path activity, since the same sample rate can be achieved at half the operating frequency. Although it has reduced activity, the blocked architecture has increased complexity and longer paths than the baseline. This results in more spread in the slack distribution, allowing more overscaling when errors can be tolerated, although errors may start at a higher voltage. Figure 5.3 shows how the power and error rate of each filter architecture vary with voltage, confirming the expected effects of the slack and activity distributions on the error rate of each architecture.

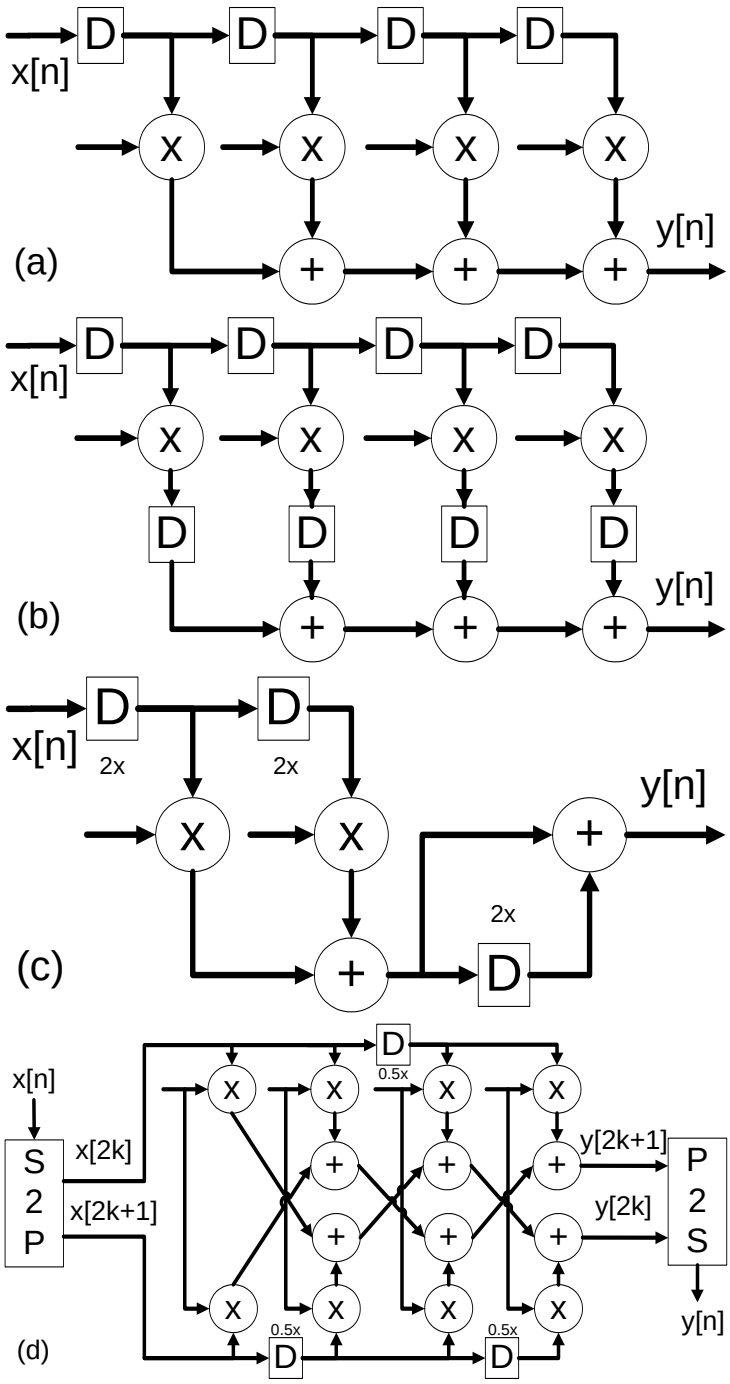


Figure 5.1: 4-tap FIR filter designs: (a) baseline, (b) pipelined, (c) folded, (d) blocked.

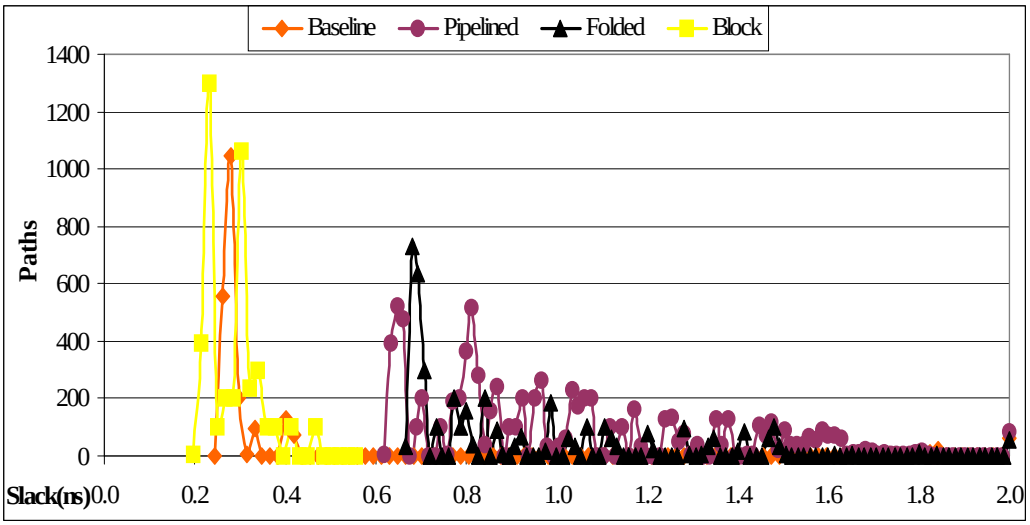


Figure 5.2: Slack distributions for the FIR filter architectures.

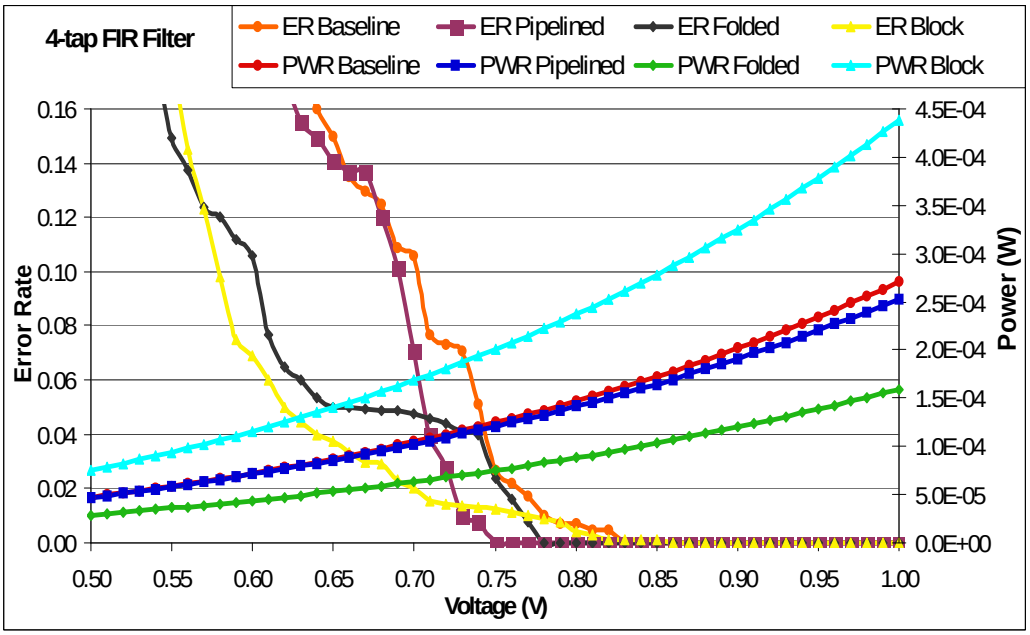


Figure 5.3: Power and error rate vs. voltage for the FIR filter architectures.

Table 5.1: Mean and standard deviation of path slack, relative to the sampling period, and average path activity normalized against the baseline.

	Baseline	Pipelined	Folded	Blocked
μ_{slack}	0.183	0.496	0.449	0.154
σ_{slack}	0.185	0.159	0.145	0.124
$Avg(\alpha_{path})$	1.0	1.9	3.4	0.5

Our simple DSP filter examples show that the architecture of a design shapes the properties of its slack and activity distributions. We now show that the same is true for general purpose processors. As demonstrated in Section 4.1.1, error rate is a function of the slack and activity distributions, and our primary goal is to use architectural optimizations to manipulate the error rate behavior of a design. Thus, we use error rate as a proxy for slack and activity. We begin by synthesizing four variants of the FabScalar [73] processor with different microarchitectural characteristics.

Figure 5.4 shows that the four different FabScalar microarchitectures have significantly different error rate behavior, demonstrating that slack, activity, and error rate indeed depend on microarchitecture. Differences in the error rate behavior of different cores are due to several factors. First, changing the sizes of microarchitectural units like queues and register files changes logic depth and delay regularity, which in turn affects the slack of many timing paths. Secondly, varying some architectural parameters such as superscalar width has a significant effect on complexity [74]. Changing complexity, fanout, and capacitance change path delay sensitivity to voltage scaling and cause the shape of the slack distribution to change. Finally, changing the architecture alters the activity distribution of the processor, since some units are stressed more heavily, depending on how the pipeline is balanced. High activity in units with many critical paths can cause error rate to increase more steeply. Likewise, an activity pattern that frequently exercises longer paths in the architecture limits overscaling. For example, long dependence chains lengthen the dynamically exercised critical paths of structures such as the issue queue and load-store queue, which perform dependence checking. As these queues become full, they begin to generate errors at higher voltages.

5.2.2 Architectural Optimizations that Manipulate Slack and Activity Distributions

Now that we understand the relationships between slack, activity, error rate, and architecture, we consider what must be done to optimize processor architecture for improved timing speculation efficiency. In this section, we propose specific architectural optimizations for general purpose processors that manipulate their slack and activity distributions. In Section 5.4, we show how

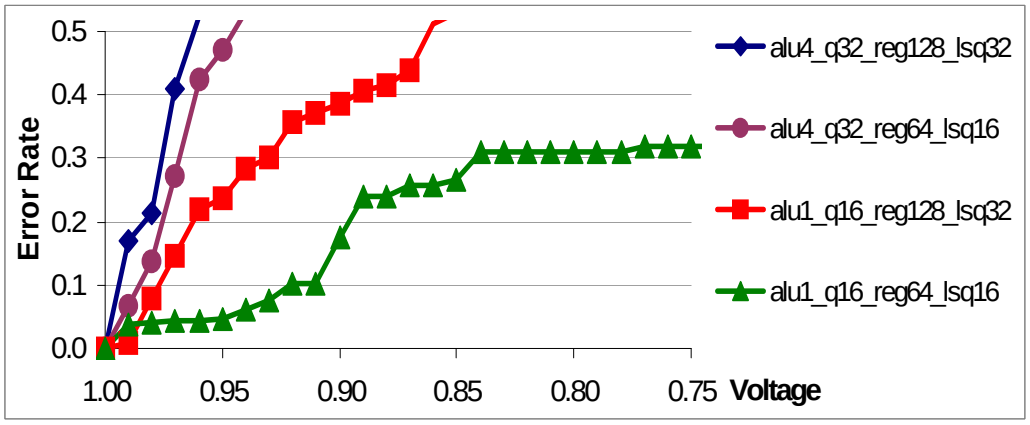


Figure 5.4: Different microarchitectures exhibit different error rate behaviors, demonstrating the potential to enhance the energy efficiency of a timing speculative architecture through microarchitectural techniques. (Notations described in Section 5.4.3.)

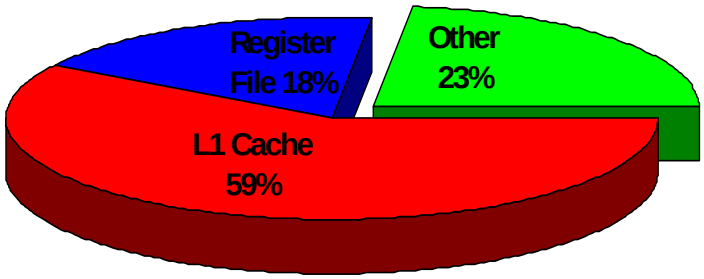


Figure 5.5: Typically, slack distributions of processors are dominated by regular structures. Caches and register files account for a large fraction of the critical paths of a processor [75].

these changes to the slack and activity distributions translate into significant energy savings for timing speculative architectures.

Regular Structures Typical energy-efficient processors devote a large fraction of die area to structures with very regular slack distributions, such as caches and register files. These structures typically have high returns in terms of energy efficiency (performance/watt) during correct operation. For example, 75–80% of the critical paths in the Alpha EV7 reside in the L1 caches and register files (Figure 5.5) [75].

While regular structures are architecturally attractive in terms of processor efficiency for correct operation, such structures have slack distributions that allow little room for overscaling, because all paths in a regular structure

Table 5.2: Mean and standard deviation of path slack, relative to the clock period, for the Alpha processor with different register file sizes.

	16reg	32reg	64reg
μ_{slack}	46%	41%	34%
σ_{slack}	10%	9%	6%

are similar in length, and when one path has negative slack, many other paths also have negative slack. For example, consider a cache. Any cache access includes the delay of accessing a cache line, all of which have nearly the same delay. So, no matter which cache line is accessed, the delay of the access path will be nearly the same. Compare this to performing an ALU operation, where the delay can depend on several factors including the input operands and the operation being performed. When exploiting timing speculation-based error resilience for energy reduction, the energy-optimal error rate is found by balancing the marginal benefit of reducing the voltage with the marginal cost of recovering from errors [19]. When many paths fail together, error rate and recovery overhead increase steeply upon overscaling, limiting the benefits of timing speculation. Reducing the number or delay of paths in a regular structure can reshape the slack distribution, enabling more overscaling and better timing speculation efficiency.

For the Alpha core, the register file is the most regular critical structure. Figure 5.6 shows slack distributions for the Alpha core with different register file sizes. As the size of the register file increases, the regularity of the slack distribution also increases, as does the average path delay. Figure 5.6 confirms that the spread of the slack distribution decreases with a larger register file. Additionally, path slack values shift toward zero (critical) slack due to the many critical paths in the register file. Table 5.2 shows standard deviation and mean values for the slack distributions of the processors with different register file sizes. The table confirms that regularity (represented by the standard deviation of slack) increases, and average slack decreases with the size of the register file. (Note that smaller σ_{slack} means a more regular slack distribution.) We confirmed similar behavior when the cache size was changed. For example, σ_{slack} reduced by 25% for the Alpha core and 23% for the MIPS core when the cache size was increased from 2 kB to 4 kB.

Architectural design decisions that reshape the slack distribution by devoting less area to regular structures or moving regular structures off the critical

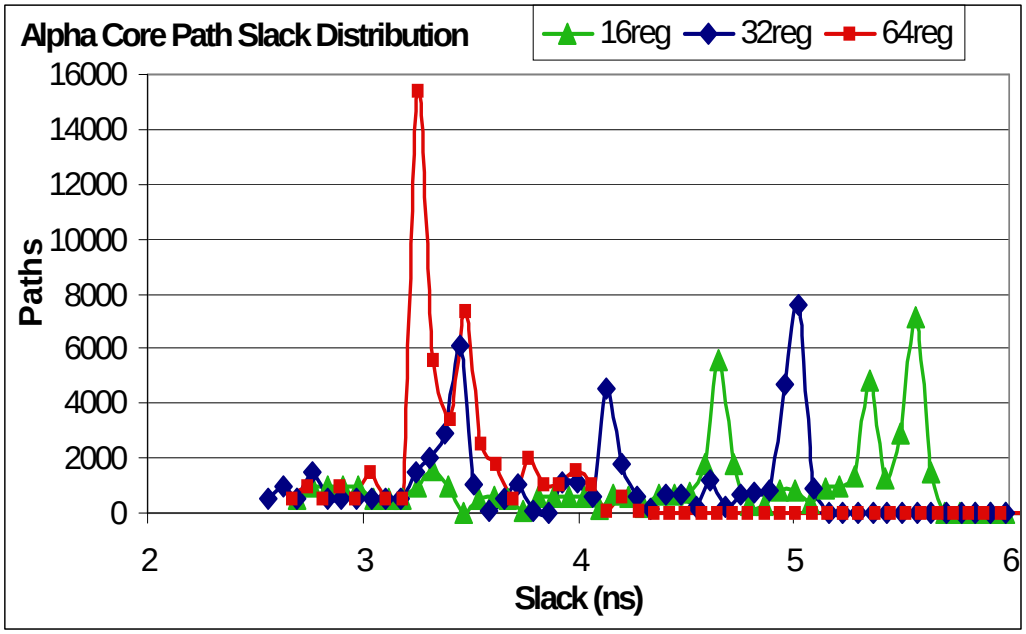


Figure 5.6: Reducing the size of the register file (a regular structure) increases the spread of the slack distribution, resulting in fewer paths bunched around the point of critical slack.

path can enable more overscaling and increase energy efficiency for timing speculative processors. In other words, additional power scaling enabled by architectures with smaller regular structures can outweigh the energy benefits of regularity when designing a timing speculative architecture. Since regularity-based decisions may also impact power density, yield, and performance, the final architectural decision should consider these constraints in addition to the optimization metric. Section 5.4 presents examples showing that reducing the regularity of the slack distribution can provide significant energy benefits when employing Razor-based timing speculation.

Note that [76] also advocates several choices that may affect the delay regularity of an architecture. However, unlike [76], our goal is not necessarily to increase slack but rather to reshape the slack and activity distributions of a processor. Decisions advocated in [76] increase slack but also make the slack distribution more regular. For example, when choosing the architecture for an arithmetic unit, we might advocate selection of a ripple-carry adder for its irregular slack distribution and lower average case delay [51], despite its higher critical path delay. [76], on the other hand, would choose a Kogge-Stone adder to decrease critical path delay, also making the slack distribution

more regular.

Logic Complexity Typically, processors are architected for energy efficiency during error-free operation at a single power and performance point and are not expected to scale to other points. However, timing speculative architectures achieve benefits by scaling beyond the typical operating point to eliminate conservative design margins. The change in the shape of the slack distribution as voltage changes depends on the delay scalability of the paths. Therefore, unlike conventional architectures, architectures optimized for timing speculation should consider the delay scalability of different microarchitectural structures.

There are several architectural characteristics that affect delay scalability that conventional processors ignore to varying degrees. One factor that affects delay sensitivity to voltage scaling is logic complexity. In a conventional processor, microarchitectural components are optimized largely oblivious to complexity, as long as the optimization improves processor efficiency at the nominal design point. However, more complex structures with more internal connections, higher fanouts, deeper logic depth, and larger capacitance are more sensitive to voltage scaling, potentially limiting overscaling for a timing speculative processor.

Figure 5.7 demonstrates how the critical path delay of the ALU of the OpenSPARC T1 [52] processor changes with voltage scaling. Path P1 is the critical path at nominal voltage. However, the delay of P2 is more sensitive to voltage scaling due to increased fanout. The slack distribution of a processor with many complex logic structures becomes more critical more quickly as voltage is scaled, limiting overscaling.

To maximize the energy efficiency benefits of timing speculation, architectural decisions should be scalability-aware. For example, complex architectural structures with high degree of fanout should be optimized to reduce complexity, if possible. Similarly, less complex implementations of architectural units can be chosen when performance is not significantly impacted. Example optimizations include changing superscalar width and queue sizes – factors that strongly influence logic complexity. The capacitance of a logic structure also influences the rate at which delay increases with voltage reduction. If the impact on processor efficiency is acceptable, less area should be devoted to complex and centralized structures with high internal capacitance

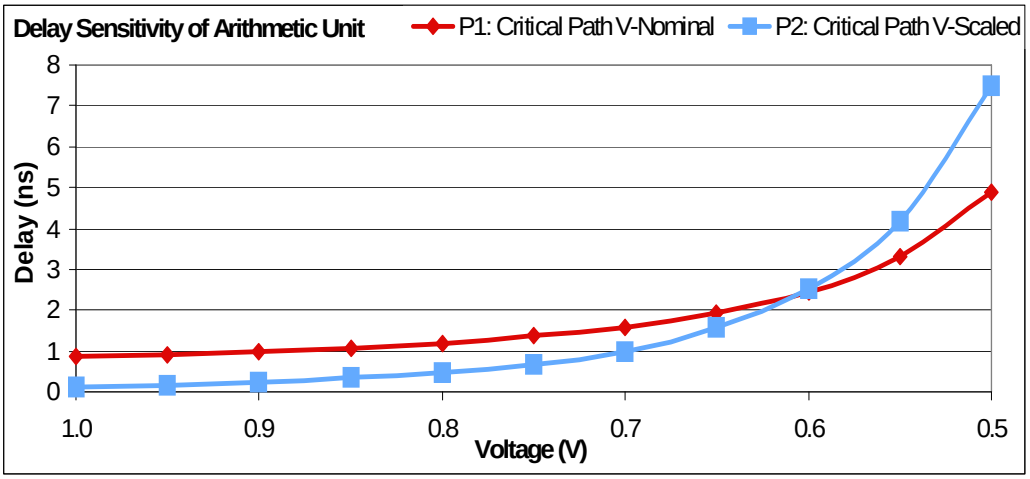


Figure 5.7: Some paths are more sensitive to voltage scaling than others. Complex logic with many high fanout paths like P2 can limit overscaling in a timing speculative architecture.

(e.g., rename logic, wakeup/select logic, bypass logic, etc.).

Comparing the Alpha and MIPS architectures reveals again how architectural changes affect the slack distribution. Figure 5.8 compares the slack distributions of the MIPS and Alpha processors. The MIPS slack distribution has both higher mean and standard deviation than the distribution for the Alpha processor, indicating reduced regularity and complexity. These factors can be attributed to reduced word length, simpler ALU design, smaller area devoted to the register file, and a simpler, smaller instruction set, which results in less complex control logic throughout the processor.

Utilization Modern processors consistently employ architectural techniques such as pipelining, superscalar processing, and caching to improve utilization by reducing the number of control and data hazards and mitigating long latency memory delays. In general, when designing for correctness, architectural design choices that increase utilization are desirable, as higher utilization of a processor core often leads to better performance. However, architectures with highly utilized critical paths are susceptible to high error rates, since increased activity on negative slack paths means more frequent errors. Architectural optimizations that reduce the activity of critical paths have the potential to reduce the error rate when timing speculation is performed.

The filter architectures described in Section 5.2.1 demonstrate how changes

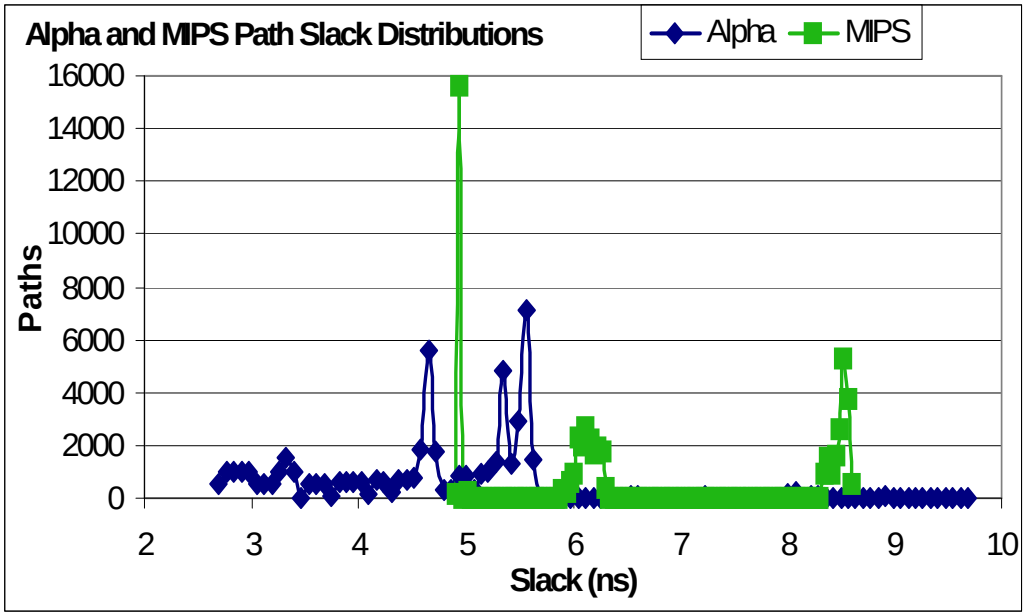


Figure 5.8: The reduced regularity and complexity of the MIPS architecture, compared to the Alpha architecture, results in a slack distribution with greater average slack and reduced regularity.

to the architecture affect the activity distribution. Table 5.1 shows that average path activity can be varied over a range of 6.8× by changing the amount of parallelism used in the filter architecture.

Superscalar Width As noted above, superscalar width has a stong impact on processor complexity [74]. In addition, changing the superscalar width can significantly impact the activity distribution of a processor. We evaluate the effect of changing the superscalar width of the MIPS architecture. We observed that average activity increases by up to 25% for the superscalar version of the processor, compared to the scalar version. Section 5.4 provides results that show how architectural changes that affect the activity distribution alter the energy efficiency of Razor-based timing speculation.

Note that activity reduction has associated costs in terms of performance during correct operation. We do not advocate reducing activity at all costs, but rather balancing the error rate reduction and energy efficiency benefits of activity reduction with the throughput benefits of high utilization. Note also that a work such as [76] is unconcerned with the activity distribution of a processor, since the goal is to *prevent* errors, not to reshape the error distribution.

Pipeline Depth The relationship between pipeline depth and energy efficiency is well understood in the context of error-free architectural design [77]. The energy-optimal pipeline depth of an architecture is reached when the marginal benefit of adding a pipeline stage equals the marginal cost, according to the performance/power relationship defined by the energy metric. The benefit of increased pipeline depth is additional timing slack, which translates into increased frequency (performance) or reduced voltage (power). The cost of increased pipeline depth is increased latch area and power, as well as reduced throughput (IPC).

While increasing the pipeline depth can result in increased energy efficiency for the zero error rate case, increasing the pipeline depth may also increase the cost of error recovery, because the cost of error recovery is proportional to the depth of the pipeline for many error recovery mechanisms [19]. Consequently, the optimal pipeline depth for an error-resilient architecture is less than the optimal depth when designing for correctness. Ignoring the overhead of error recovery for an error-resilient architecture can result in selection of a suboptimal pipeline depth.

We formulate an expression for the optimal pipeline depth for an error-resilient architecture by modifying Hartstein and Puzak's model for optimal pipeline depth [77]. The model combines expressions for performance and power to produce an energy efficiency metric (performance/power). Optimal pipeline depth is found by maximizing the metric. We modify the power and performance expressions of the original model to account for the effects of error-resilient design and operation on the pipeline. As such, the power and delay expressions are modified to incorporate the effects of voltage scaling, and the performance equation is modified to include the penalty of stalling to correct errors, according to the operating voltage and resulting error rate. Equation 5.1 gives the performance expression for the updated model, Equation 5.2 gives the power expression, and Equation 5.3 combines the expressions to form the energy efficiency metric. Table 5.3 explains the meaning of each model parameter.

$$\frac{T}{N_I} = \frac{1}{f_s a} + \frac{\gamma_h N_h p}{f_s} + \frac{\gamma_e e p T_o}{N_I} \tag{5.1}$$

Table 5.3: Parameters for the pipeline energy efficiency model.

T	Time
N_I	Number of instructions
f_s	Frequency ($f_s = 1/(t_0 + t_p/p)$)
t_0	Latch delay
t_p	Logic delay of the pipeline
p	Pipeline depth
a	Average degree of superscalar processing
γ_h	Hazard recovery time as a fraction of pipeline delay
N_h	Number of hazards
γ_e	Error recovery time as a fraction of pipeline delay
e	Error rate (error cycles/ total cycles)
P_T	Total power
f_{cg}	Clock gating factor
P_d	Dynamic power
P_l	Leakage power
N_L	Number of latches
η	Latch growth factor
f_v	Voltage scaling factor
v_o	Normalized critical voltage
k	Regularity factor (relates path slack to pipeline depth)
w	Criticality factor (relates error acceleration to voltage)

$$P_T = (f_{cg} f_s P_d f_v^2 + P_l f_v) N_L p^\eta \tag{5.2}$$

$$BIPS^m/W = ((T/N_I)^m P_T)^{-1} \tag{5.3}$$

The equation describing the performance of an error-resilient architecture (Equation 5.1) includes an additional term ($\gamma_e e p T_o / N_I$) to model the relationship between pipeline depth and error recovery overhead. To model the impact of voltage overscaling on processor power and reliability, we introduce a voltage overscaling factor (f_v). Dynamic power scales quadratically with voltage, and leakage power scales linearly with voltage. The voltage scaling factor also influences the error rate, since path delays increase as voltage decreases. Equation 5.4 describes how the error rate increases as voltage is scaled down. The error acceleration parameter (w) describes the rate at

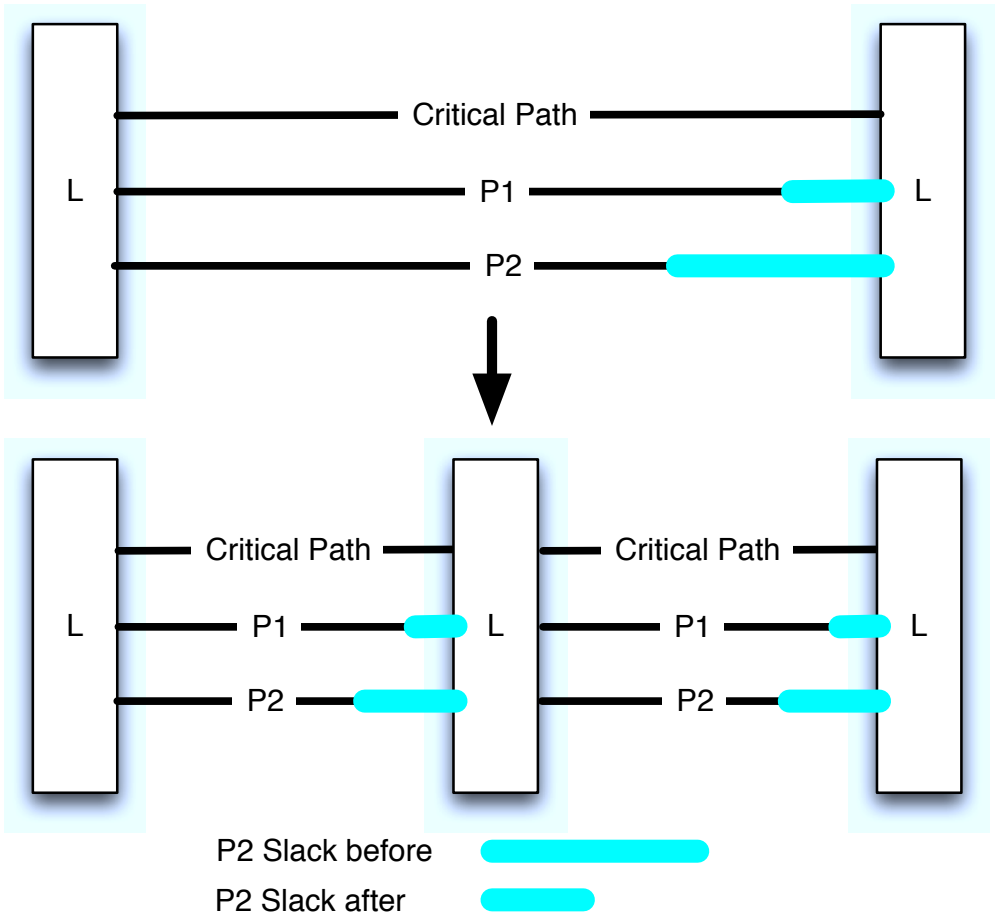


Figure 5.9: Pipelining alters the slack distribution. The highlighted segments denote path slack. When pipeline depth increases, the lengths of the timing paths and the amount of timing slack per stage are reduced.

which error rate increases after scaling past the critical voltage (v_o).

$$e = min(1, ((1 - f_v)/(1 - v_o))^w) \tag{5.4}$$

The critical voltage (v_o) depends on pipeline depth as well. This is because adding pipeline stages reduces not only the delay of each stage but also the timing slack of each stage. Figure 5.9 illustrates this effect. Equation 5.5 models the dependence of v_o on the length of the pipeline. In the equation, v_{ob} denotes the normalized critical voltage for the baseline pipeline, with depth p_b. (We assume a traditional 5-stage pipeline as the baseline.) The regularity factor (k) controls how quickly the number of negative slack paths

(and thus error rate) grows with the number of pipeline stages. As the pipeline depth grows larger than the baseline pipeline depth, the amount of available timing slack decreases proportionally. Note that the equation assumes that pipelining divides all timing paths equally. All previous works on optimal pipeline depth make the same assumption.

$$v_o = 1 - (1 - v_{ob}) * (p_b/p)^k \tag{5.5}$$

The model described above was used to evaluate the energy efficiency of a processor architecture at different error rates and pipeline depths, in order to find the dependence of optimal pipeline depth on an error resilience mechanism. Each error resilience mechanism has a different optimal error rate. This implies that each error resilience mechanism may also have a different optimal pipeline depth. Figure 5.10 shows how energy efficiency varies with pipeline depth for architectures operating at different error rates. Each error rate represents a different magnitude of power savings and a different error recovery overhead. Notice that the optimal pipeline depth and energy efficiency vary significantly depending on the error rate, demonstrating that it is essential to take the error resilience mechanism into account when selecting the pipeline depth of an error-resilient architecture. In practice, the actual number of pipeline stages should be chosen not only based on the above considerations, but also based on the desired performance and power targets for nominal, error-free operation.

Different error resilience mechanisms have different error recovery overheads. Figure 5.11 shows the energy efficiency (normalized to the error-free baseline), as well as the optimal pipeline depth and error rate for different values of error recovery overhead (γ_e). As the recovery overhead increases, the optimal error rate decreases. Thus, the optimal pipeline depth increases. The data demonstrate that the optimal pipeline depth depends on the error recovery overhead for a given error recovery mechanism, stressing the importance of taking the error recovery mechanism into account when selecting the pipeline depth of an error-resilient architecture.

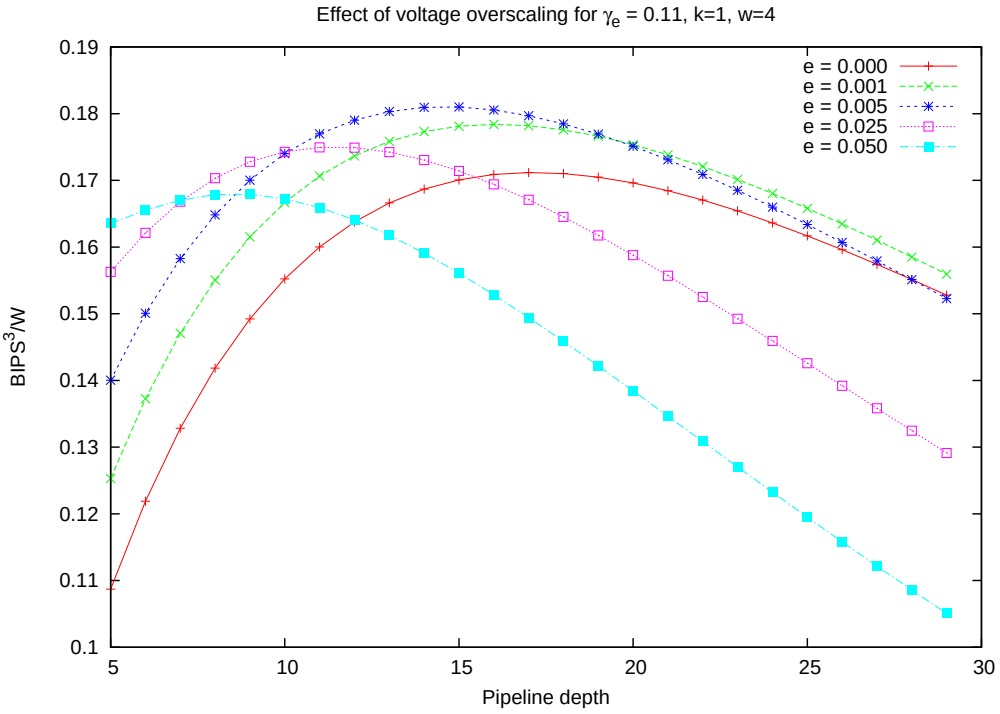


Figure 5.10: Each curve shows how energy efficiency varies with pipeline depth for a given error rate. The optimal pipeline depth varies significantly, depending on the error rate.

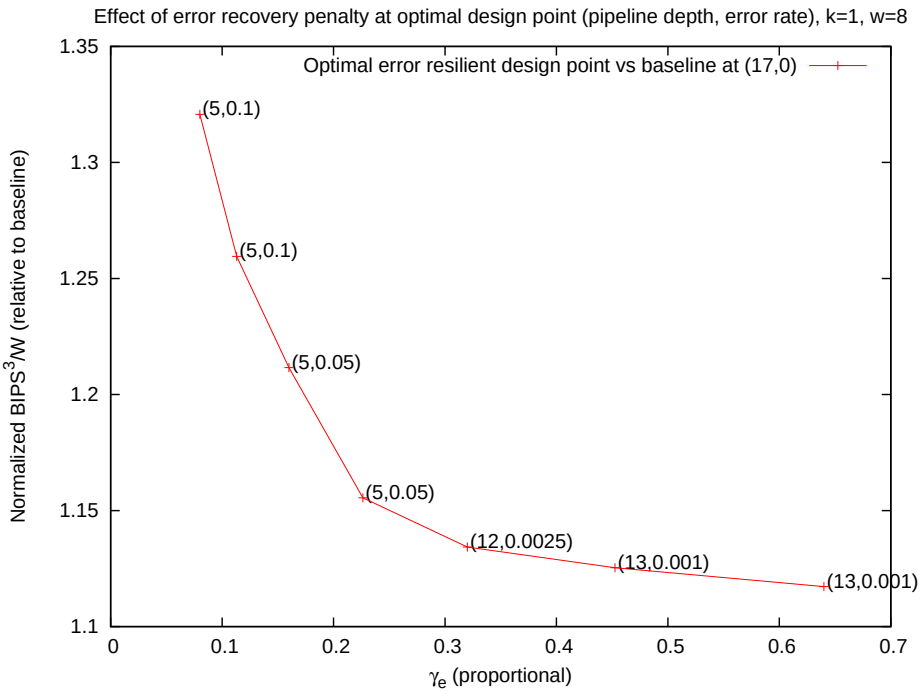


Figure 5.11: The energy-optimal pipeline depth and error rate for an architecture depend on the error recovery overhead (γ_e).

5.3 Methodology

We have developed a design flow that takes an RTL design through synthesis, placement, routing, power estimation, timing analysis, area estimation, gate-level simulation, and error rate measurement. Designs are implemented with the TSMC 65GP library (65 nm), using Synopsys Design Compiler [37] for synthesis and Cadence SoC Encounter [38] for layout. In order to evaluate the power and performance of designs at different voltages and to provide V_{th} sizing options for synthesis, Cadence Library Characterizer [78] was used to generate low, nominal, and high V_{th} libraries at each voltage (V_{dd}) between 1.0 V and 0.5 V at 0.01 V intervals. Power, area, and timing analyses are performed in Synopsys PrimeTime [53].

Gate-level simulation is performed with Cadence NC-Verilog [47] to gather activity information for the design, which is subsequently used for dynamic power estimation and error rate measurement. Please refer to Section 4.2.4 for a detailed description of error rate measurement.

In addition to inducing timing errors by increasing logic delays, voltage

scaling may prompt reliability concerns for SRAM structures, such as insufficient static noise margin (SNM). Fortunately, the minimum energy voltage for our processors is around 750 mV, while production-grade SRAMs have been reported to operate reliably at voltages as low as 700 mV [79]. Research prototypes have been reported to work for even lower voltages. In any case, modern processors typically employ a "split rail" design approach, with SRAMs operating at the lowest safe voltage for a given frequency [58].

In our evaluation of general purpose processor architectures, we run instruction traces from a set of eight SPEC benchmarks (ammp, art, equake, mcf, parser, swim, twolf, wupwise) on the processors. The traces are captured after fast-forwarding the benchmarks to their early Simpoints [59].

We model Razor-based error resilience in our evaluations (though our design principles are generally applicable to any timing speculative architecture). Table 5.4 summarizes the average processor-wide static and dynamic overheads incurred by our designs that use Razor for error detection and correction. In our design flow, we measure the percentage of die area devoted to sequential elements as well as the timing slack (with respect to the shadow latch clock skew of 1/2 cycle) of any short paths that need hold buffering. When evaluating energy at the architecture level, we account for the increased area and power of Razor flip-flops, hold buffering on short paths, and implementation of the recovery mechanism. Most of the static overhead is due to Razor FFs. Buffering overhead is small, and the availability of cells with high and low V_{th} provides more control over path delay, eliminating the need for buffering on most paths. We also add energy and throughput overheads proportional to the error rate to account for the dynamic cost of correcting errors over multiple cycles. We model a counterflow pipeline Razor implementation [19] with correction overhead proportional to the number of processor pipeline stages (P). We conservatively replace all sequential cells with Razor FFs. This conservative accounting measure means we can also claim greater immunity to aging-induced errors, e.g., due to NBTI, which can cause paths to become critical over time.

To evaluate the effects of architectural optimizations on the energy efficiency of timing speculation, we perform an exploration of the processor design space defined by the parameters found in Table 5.5. All other parameters were chosen to be identical to the OpenSPARC core. Because it would be unreasonable to write, synthesize, layout, and test custom RTL for each

Table 5.4: Average processor-wide Razor overheads for error-tolerant architectures.

Hold buffering	Razor FF	Counterflow	Error Recovery
2% energy	23% energy	<1% energy	P cycles

Table 5.5: Design parameters and their possible values.

I/D$ kB	ALU/FPU	INT Q-FP Q	INT/FP Regs	Ld/St Q
4,8,16,32	1,2,4	32-16,64-32	64,128	32,64

of the hundreds of OpenSPARC processor configurations that we study, we instead evaluate the power, performance, and error rate of the architectures using a combination of gate and microarchitecture-level simulation.

To estimate the performance and power of different architectures, we use SMTSIM [55] with Wattch [56]. We also use Wattch to report the activity factor for each microarchitectural structure in each configuration, for each benchmark. We approximate the error rate of an architecture as the weighted sum of error rates from each of the microarchitectural components that we vary in our exploration. To obtain the component error rates, we used RTL from the OpenSPARC T1 processor [52]. We modified the existing OpenSPARC module descriptions to create an RTL description for each component configuration in Table 5.5 and used our detailed design flow, as described above, to measure error rate and power at different voltages. Error rate at the architecture level is given by the sum of the component error rates, where each component error rate is weighted by the activity factor captured during architecture-level simulation. While this error rate estimation technique is not as accurate as our design-level technique, it provides suitable accuracy to study the error behavior of many architectures without requiring full gate-level evaluations of many complex architectures.

5.4 Experimental Results

In Section 4.1.1, we showed how the slack and activity distributions determine the error rate. In Section 5.2, we showed how architecture influences the slack and activity distributions. In this section, we demonstrate that architectural optimizations can significantly improve the energy efficiency of

timing speculation, first for simple DSP filter architectures, and then for general purpose processor cores (Alpha, MIPS, and OpenSPARC).

5.4.1 DSP Filter Architectures

First, we compare the filters with respect to different energy efficiency metrics over a range of error rates to observe how the optimal architecture changes for error-free and error-resilient operation. Figure 5.12 compares the filter architectures in terms of power-delay product. The low capacitance, shorter paths, and highly regular slack distribution of the pipelined architecture allow it to achieve better energy efficiency for error-free operation. However, the clustering of path delays in the pipelined design causes the error rate to increase rapidly once errors begin to occur. Thus, power savings quickly level off for the pipelined architecture. Consequently, the blocked architecture becomes more energy efficient at moderate error rates. While higher complexity and deeper logic depth limit the amount of voltage scaling for correct operation with the blocked architecture, low activity allows the error rate of the filter to stay lower longer as voltage is reduced, enabling an extended range of power savings for the blocked design. The baseline and folded architectures do not minimize energy over any range of error rates, due to the high activity and regularity of the folded architecture and the increased sensitivity to voltage scaling of the baseline (without the benefit of reduced activity that the blocked architecture has).

The choice of the efficiency metric (which expresses the relative importance of power and performance to the architect) influences which architecture is most efficient at different error rates. Figure 5.13 compares the filters in terms of power efficiency. Both the pipelined and folded architectures are approximately the same in terms of sensitivity to voltage scaling and regularity. The pipelined filter has the best power efficiency for low error rates, due to extra slack afforded by the increased regularity of the slack distribution. This extra slack enables more scalability before the onset of errors. However, regularity results in a steep increase in the error rate, allowing the folded architecture to gain the power efficiency edge for mid-range error rates. The folded architecture has reduced complexity, fewer paths, and less fanout, resulting in the best scalability of any architecture. It also has low power

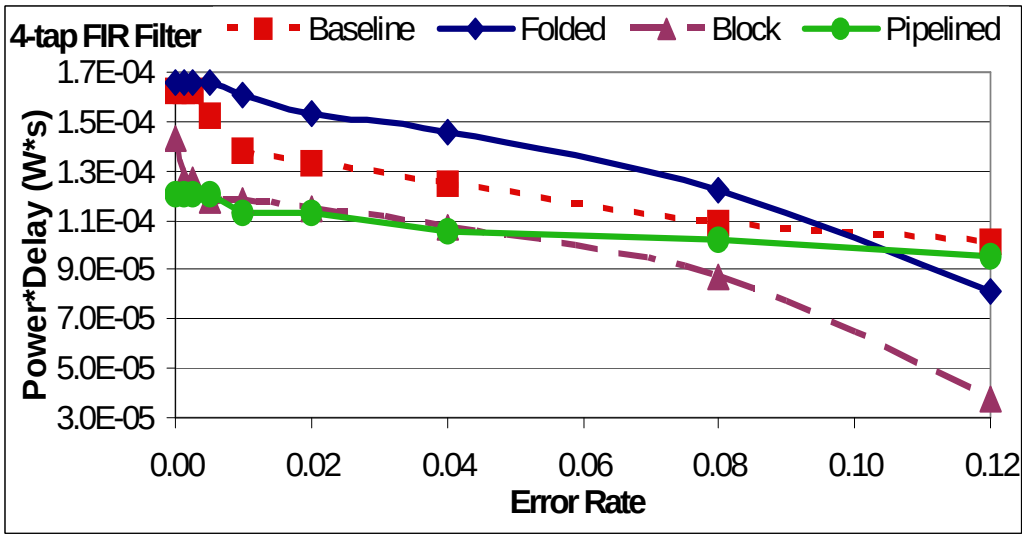


Figure 5.12: Energy efficiency comparison showing crossovers between filter architectures for different voltage overscaling-induced error rates.

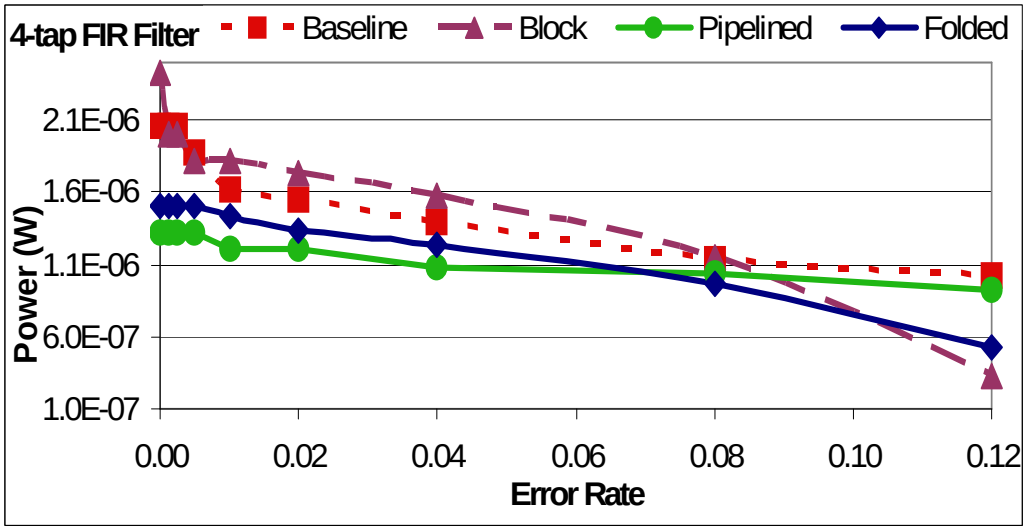


Figure 5.13: Power efficiency comparison showing crossovers between filter architectures for different voltage overscaling-induced error rates.

consumption due to simple logic and low area (Figure 5.3). Nevertheless, though it has better scalability and low power, once it starts making errors, its error rate increases dramatically, due to increased activity. This behavior allows the block filter, with reduced activity, to take the lead at high error rates.

Figure 5.14 compares the energy consumption of Razor implementations of the filter architectures. While the pipelined architecture has the best energy efficiency for error-free operation, the blocked architecture consumes the least energy for Razor-based timing speculation (29% less energy than the error-free pipelined filter). The reduced activity of the blocked filter allows more voltage scaling before the energy-optimal error rate for Razor is reached. Furthermore, the blocked filter, having fewer flip-flops and pipeline stages, has reduced implementation and recovery overheads for Razor, making it a more efficient choice for exploiting error resilience. Note that other filter architectures, including the optimal architecture for correct operation, do not achieve energy reduction with Razor, either due to static overheads (Razor flip-flops and buffering of short paths) or dynamic overheads (power and energy costs of error recovery). This result demonstrates the importance of timing speculation-aware architectural optimization techniques.

To summarize, our experiments with different DSP filter architectures validate our claim that the optimal architecture for correctness may not be efficient for exploiting timing error resilience. The results also confirm that architectural optimizations that alter the slack and activity distributions have the potential to increase the energy efficiency of timing speculation.

5.4.2 General Purpose Processor Architectures

In this section, we evaluate how changes to Alpha, MIPS, and FabScalar architectures that affect their slack and activity distributions (as described in Section 5.2.2) influence their energy efficiency for timing speculation. Figure 5.15 compares the energy efficiency of the Alpha processor for varying register file sizes. The design with a larger register file has higher throughput and better energy efficiency when both processors operate error-free. However, the higher average path delay and path delay regularity associated with the larger register file hinder voltage scaling and energy efficiency at non-zero

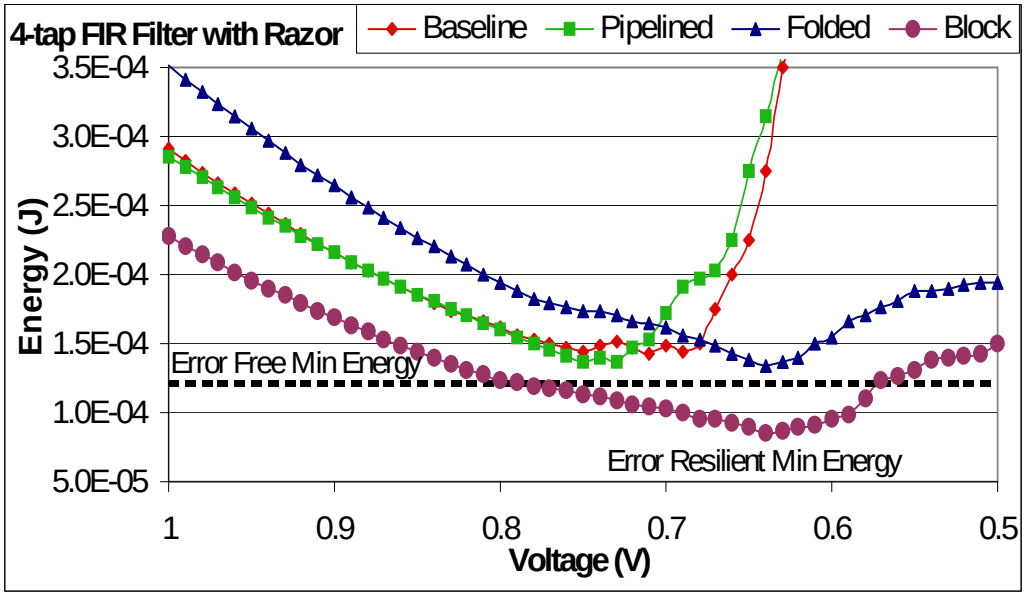


Figure 5.14: Minimum energy for correct operation (denoted by the dotted line) is achieved with the pipelined architecture. When Razor is used to enable timing speculation, the blocked architecture minimizes energy, demonstrating that the architecture that minimizes energy by exploiting error resilience is different than the optimal architecture for error-free operation.

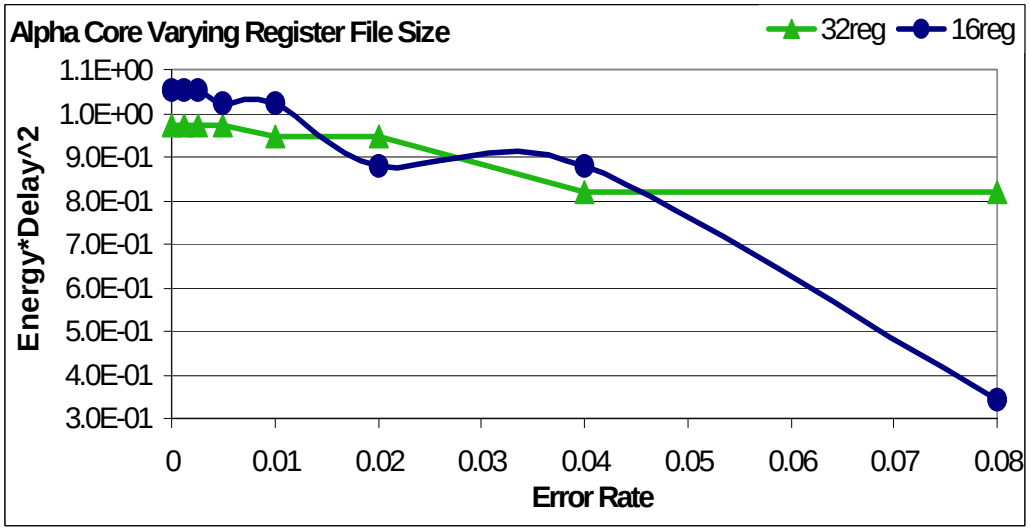


Figure 5.15: A larger register file increases performance, but also results in increased regularity and activity, hindering voltage scaling and energy efficiency at larger error rates.

error rates. Furthermore, high performance corresponds to higher activity, which causes error rate to increase more quickly for the processor with the larger register file.

Because of the higher throughput of the 32-register design, there is a small range of error rates over which the 32-register design regains the efficiency advantage when the many regular paths in the 16-entry register file begin to have negative slack, and error rate begins to increase more rapidly. However, the design with fewer registers is able to scale to a much lower voltage for higher error rates because of its lower activity, increased average slack, and more gradually increasing error rate resulting from reduced regularity of the slack distribution.

Figure 5.16 shows energy consumption for the Alpha core with Razor-based timing speculation, confirming that the architecture with a smaller register file exploits timing error resilience more efficiently. The 16-register architecture reduces energy by 21% with respect to the optimal architecture for correctness, while the optimal error-free architecture barely procures any energy savings (2%) when using Razor. Again, we observe significantly improved benefits from optimizing the architecture to exploit timing error resilience while the optimal error-free architecture sees only a small energy reduction with timing speculation.

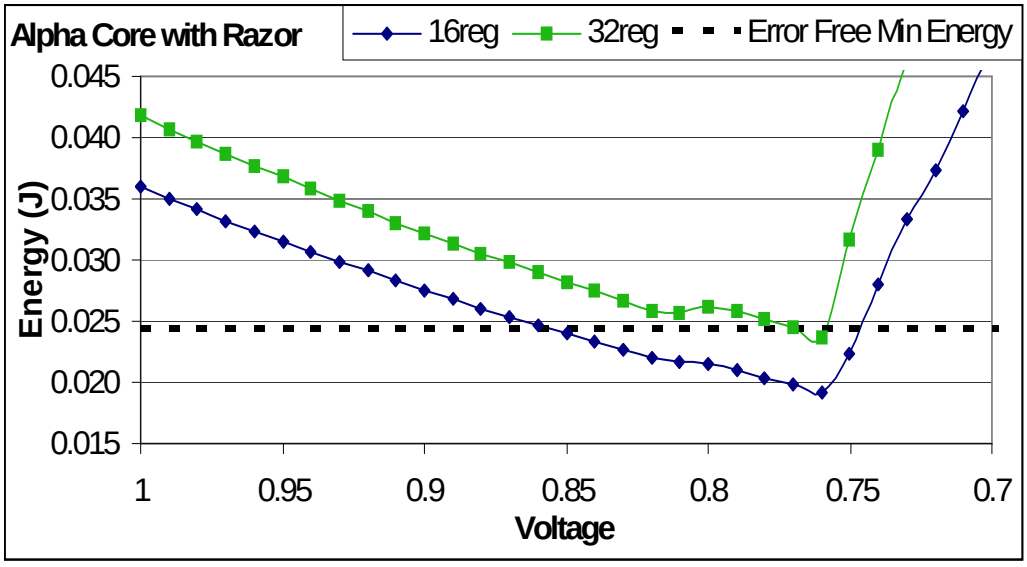


Figure 5.16: The 16-register design, having reduced regularity and activity, achieves significant energy savings with Razor, while the 32-register design, which was optimal for correct operation, achieves almost no benefit.

We evaluated the energy efficiency of the MIPS processor at different error rates when the superscalar width (and number of ALUs) was increased. The main effect on the error rate from increasing the superscalar width of the processor is due to increased activity. Not only does this architectural change increase the throughput (and thus the activity factor) of the processor, increasing the superscalar width also increases the number of paths that are active when the processor is able to exploit ILP on multiple ALUs.

Figure 5.17 compares a single-ALU version of the MIPS architecture against one with two ALUs. The multiple-ALU architecture has better energy efficiency for correct operation due to increased throughput (up to 21% throughput reduction for the scalar case, 13% on average). However, when operating at non-zero error rates, the increased activity and complexity of the multiple-ALU architecture causes the error rate to increase more rapidly, limiting voltage scaling for higher error rates. More instructions per cycle means more errors per cycle, and more active ALUs means more paths causing errors when voltage is scaled down. The higher activity of the multiple-ALU architecture makes the single-ALU architecture more energy-efficient for most non-zero error rates.

Figure 5.18 confirms that the scalar design exploits timing error resilience

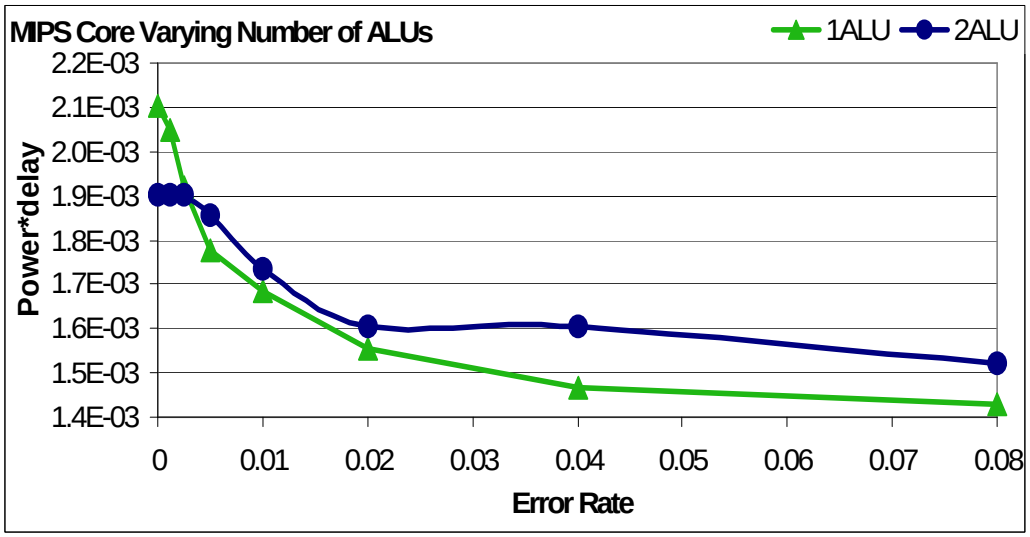


Figure 5.17: Increased throughput for the multiple-ALU architecture results in better energy efficiency for error-free operation, but increased activity results in worse efficiency at most non-zero error rates.

more efficiently. Whereas the superscalar pipeline achieves better energy efficiency for correct operation, increased complexity and activity, along with increased implementation and recovery overheads for Razor, prevent the multiple-ALU architecture from achieving energy benefits with Razor. The single-ALU architecture has a more gradually increasing error rate, allowing extended voltage scalability and an 18% energy reduction with respect to the energy-optimal architecture for correctness.

To evaluate the potential benefits of manipulating the pipeline depth of an error-resilient processor, we would like to explore optimal pipelining for an entire processor core. However, writing RTL for an entire processor for different pipeline depths is a challenging and time-consuming task. As far as we know, no open source processor RTL exists in which the pipeline depth can be scaled arbitrarily. The closest approximation we found is FabScalar, in which certain pipeline stages can be subdivided into multiple stages. We evaluate the effects of manipulating the pipeline depth in an error-resilient FabScalar processor by comparing versions of the pipeline with issue depths 1 and 2. The issue stage has by far the most critical paths in the FabScalar processor.

Figure 5.19 compares the energy efficiency of the pipelines with issue depth (ID) 1 and 2 at different error rates. For correct operation, the ID 2 pipeline

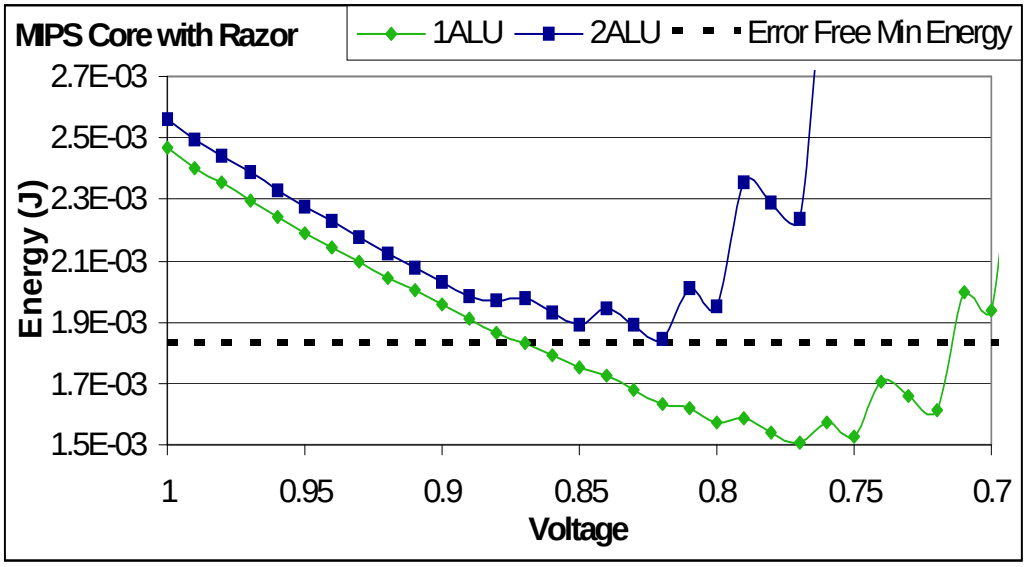


Figure 5.18: The more complex superscalar architecture has throughput and energy benefits for error-free operation but fails to achieve any energy savings with Razor-based timing speculation. The simpler, scalar design achieves substantial energy savings with Razor.

has 9% better energy efficiency, because increasing the pipeline depth of the issue stage allows the pipeline to be optimized for a higher frequency (or lower voltage), and achieve higher throughput (or lower power). As voltage is scaled down, however, the error rate of the ID 2 pipeline increases more quickly. This is because dividing a path with a pipeline latch not only partitions its logic between two stages, it also partitions the path's timing slack between two stages. Thus, pipelining reduces the average amount of timing slack in the pipelined stages, so that more paths fail sooner when voltage is scaled down. Due to the steeper increase of error rate in the ID 2 pipeline, the ID 1 pipeline has better energy efficiency at higher error rates.

Figure 5.20 compares the energy of the ID 1 and ID 2 pipelines with Razor. The ID 1 pipeline consumes 13% less energy with Razor than the ID 2 pipeline. This is due to two factors. First, as discussed above, the error rate of the ID 2 pipeline increases faster as voltage is scaled down, resulting in less voltage overscaling when Razor is used. Second, the ID 2 pipeline has a higher average error recovery cost, due to the increased average cost of pipeline flushing during error correction. These results confirm that ignoring the error resilience mechanism when selecting the pipeline depth of an error-

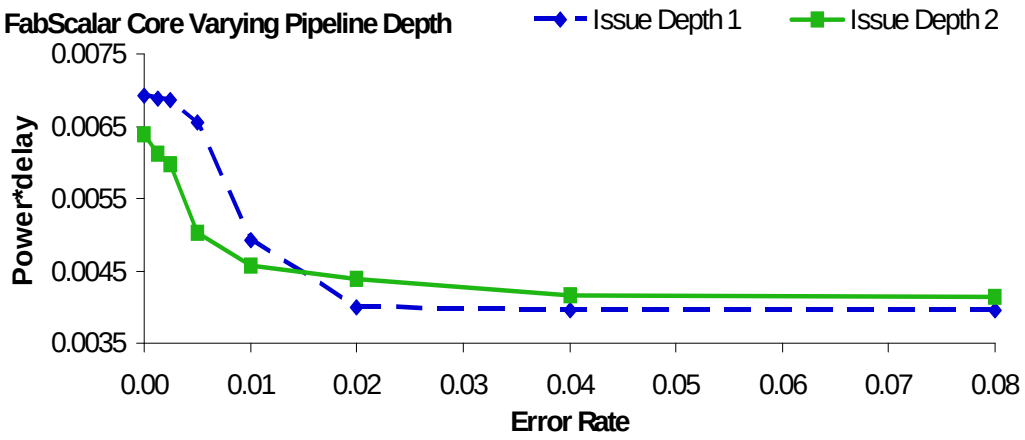


Figure 5.19: Increased pipeline depth enables the ID 2 pipeline to achieve higher energy efficiency for correct operation. However, increased pipeline depth causes the error rate to increase more quickly, and the ID 1 pipeline has better energy efficiency for higher error rates.

resilient processor can lead to energy inefficiency. An error-resilient processor should use a shallower pipeline depth than a processor that does not exploit error resilience. We expect the potential benefits of optimizing the pipeline depth to increase with more flexibility in the available pipeline depth.

To summarize, our experimental results with Alpha, MIPS, and FabScalar cores further confirm the benefits of architecting to exploit timing error resilience and demonstrate that architectures that have been optimized for energy-efficient error-free operation see little or no energy benefits when exploiting timing speculation. These results re-confirm that changing the slack and activity distributions with architectural optimizations can improve the energy efficiency of timing speculation.

5.4.3 Design Space Exploration for OpenSPARC

In the previous section, we performed analyses of various architectural optimizations to validate our insights on resilience-optimized architectures. In this section, we present an exploration of the design space for resilience-optimized general purpose processor architectures to further confirm that the benefits of exploiting error resilience can be significantly enhanced by optimizing the architecture for timing speculation.

In our exploration, we evaluated nearly 400 architectural configurations by

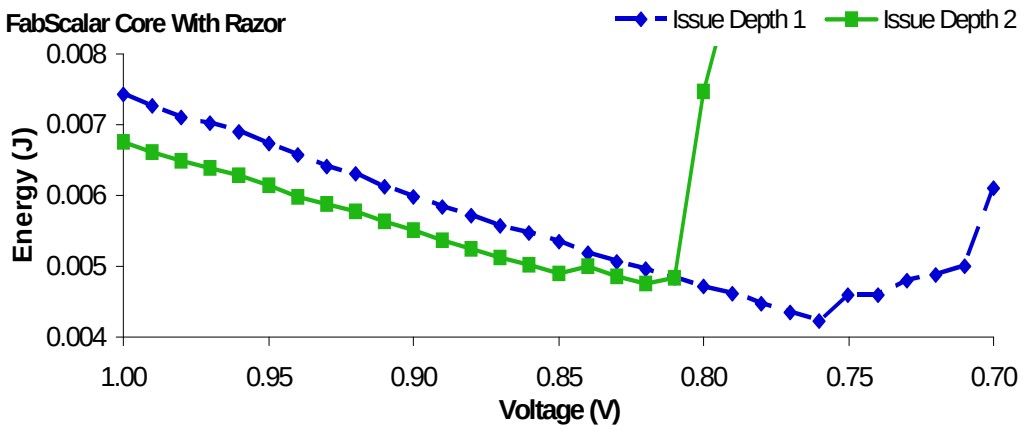


Figure 5.20: The shallower pipeline (ID 1) achieves better energy efficiency with Razor, due to lower error recovery overhead and more gradual error rate increase, afforded by increased slack.

varying instruction and data cache sizes (ic, dc), the number of integer and floating point functional units (alu), instruction queue size (q), the number of physical registers (reg), and the size of the load-store queue (lsq). A tuple (ic, dc, alu, q, reg, lsq) denotes the parameters of a particular architecture of interest. For each architecture, we estimated power, performance, and error rate as described in Section 5.3 and used these data to characterize energy consumption of the architectures at different error rates.

Figure 5.21 compares the energy efficiency of three architectures that emerged as the optimal design points for different ranges of error rates. The optimal architecture for error-free operation (ic8, dc16, alu1, q32, reg128, lsq64) has a moderate instruction cache size, larger data cache, and maximum sizes for queues and register files. For error-free operation, this configuration achieves good performance and has low power, making it the energy-optimal architecture. However, the large cache and register file sizes result in a highly regular slack distribution, so that many paths fail in groups as voltage is scaled. The increased complexity and deeper logic of large instruction and load-store queues, while increasing performance, also makes the architecture fail sooner with overscaling.

For low to mid-range error rates, a different energy-optimal architecture (ic8, dc8, alu1, q32, reg128, lsq32) emerges. This architecture has a smaller data cache and load-store queue, resulting in reduced regularity and complexity. The immediate effect of increased spread and average slack in the

slack distribution is that voltage can be scaled further before the error rate begins to increase dramatically, resulting in more power savings for timing speculation before reaching an energy-optimal error rate. When operating at low to mid-range error rates, the resilience-optimized architecture has 6% energy (W/IPC) benefits over the optimal error-free architecture. Energy reduction is mainly due to enhanced power scaling (15% power reduction, on average), since throughput is reduced by 7% with respect to the optimal error-free architecture. Thus, energy benefits will increase for a metric that weights power more heavily.

Note that compared to the optimal error-free architecture, the optimal for low to mid-range error rates decreases the size of the load-store queue (LSQ), but not the instruction queue. This is primarily because the LSQ becomes full more often than the IQ, resulting in a longer dynamic critical path that limits voltage scaling. To a second degree, the size of the instruction queue also has a more pronounced effect on performance.

For higher error rates (around 6% and up), an architecture with minimum-sized data cache and register file (ic8, dc4, alu1, q16, reg64, lsq32) consumes the least energy. In addition to the significantly reduced regularity of the slack distribution (reduced area devoted to regular structures and reduced criticality of regular structures), this architecture also has small queue sizes with decreased complexity and better scalability. The throughput of this architecture is an additional 27% lower than the correctness-optimized baseline; however, the corresponding reduced activity actually has some benefit in terms of energy, since it results in a more gradually increasing error rate as voltage is reduced. The optimal architecture for higher error rates has the most gradually increasing error rate, enabling significant voltage scaling and an average of 38% energy reduction at higher error rates with respect to the optimal error-free architecture.

Graceful failure in the presence of overscaling translates into a lower dynamic energy overhead when exploiting Razor-based timing speculation. Figure 5.22 echoes the results of our previous experiments, showing that the optimal architecture for correctness achieves only minor (5%) energy benefits with Razor, while the resilience-optimized architecture reduces energy by 25% with respect to the error-free minimum energy.

Since resilience-optimized architectures typically reduce the sizes of regular structures like caches and use simpler architectural features that may throttle

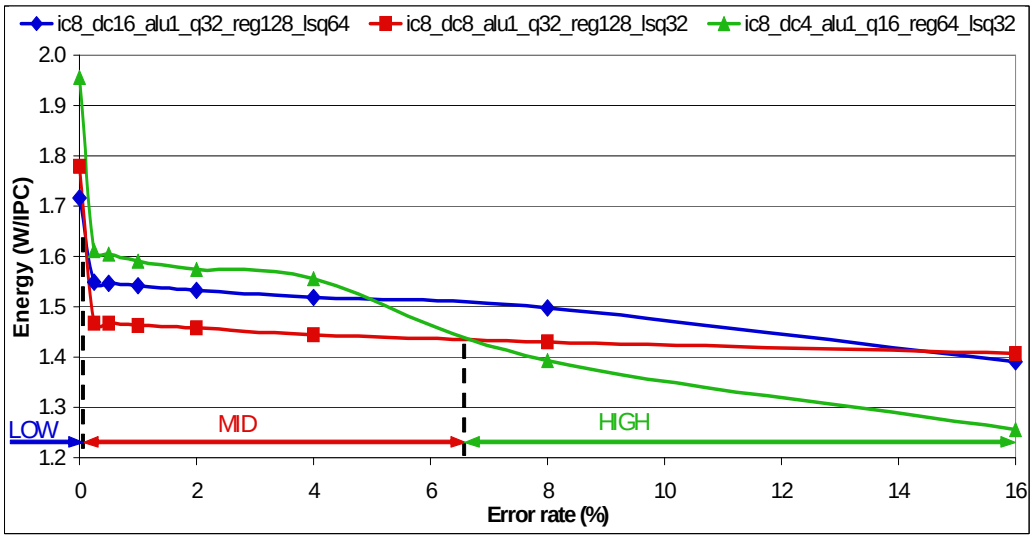


Figure 5.21: The energy-optimal architecture is different for different ranges of voltage overscaling-induced error rates.

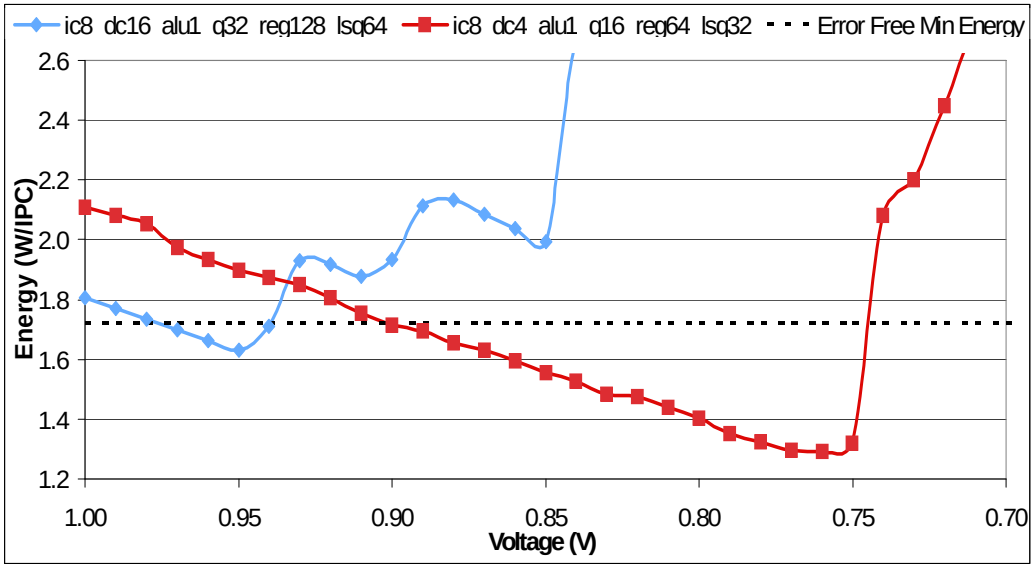


Figure 5.22: The resilience-optimized architecture achieves significant energy savings with Razor, while the optimal error-free architecture sees only minor benefits.

Table 5.6: Throughput reduction for resilience-aware optimizations.

REG (32 → 16)	ALU (2 → 1)	OpenSPARC
6%	21%	27%

ILP, they may sacrifice some throughput in order to reduce energy. Table 5.6 shows throughput reduction for the resilience-aware optimizations we evaluated in this section. Since we employed voltage overscaling, we demonstrate power and energy savings at the expense of some throughput. Note, however, that we could also demonstrate performance gains by overscaling frequency rather than voltage.

5.5 Related Work

The slack and activity distributions of a processor influence the error rate and the efficiency of timing speculation. Furthermore, architectural optimizations have the potential to alter the slack and activity distributions to increase the energy efficiency of timing speculation.

Some related work [80] proposes using different arithmetic architectures for stochastic computing, enabled by changing the representation of data. Building upon von Neumann's formulation of stochastic computing (discussed in Section 2.1) that uses stochastic bitstrings to represent numbers, [80] observes that operations like multiplication and addition can be performed approximately with much simpler binary logic when variables are encoded as probabilities. However, such formulations simply perform approximate computing on deterministic logic and miss one of the fundamental drivers for our vision of stochastic computing – the fact that hardware is inherently non-deterministic and faking determinism is expensive.

A related body of work, discussed in Chapter 4, exists at the level of design techniques that optimize circuit modules for a target error rate [48] or to fail gracefully in response to voltage overscaling [32, 33] through cell-based optimizations. Whereas these design-level techniques reshape the slack distribution or reliability of a circuit module, the architecture-level techniques presented in this chapter target both the slack and activity distributions of a processor. Also, architecture-level optimizations can have a greater impact on the slack distribution of a processor, since for a design-level technique,

the microarchitecture and synthesized netlist are fixed, and the ability of cell sizing to reshape path slack may be limited. A promising direction of work is to investigate co-optimization at the architecture and design levels to reshape the slack and activity distributions and maximize the energy efficiency benefits provided at each level.

Another relevant related work [76] explores microarchitectural parameter selection to optimize processor performance in the presence of process variations. The authors aim to reduce performance loss due to process variations by adding slack to the critical paths of a processor where possible. However, unlike our work, [76] attempts to *prevent* the onset of errors; they are not concerned with the activity distribution of the processor or scalability *after* the point where errors begin to occur. Our work, on the other hand, focuses on the error rate distribution. Since the authors of [76] are only concerned with correct operation, they have no reason to consider the activity distribution of a processor or the shape of the slack distribution. We consider all these factors in our approach to architecture, since they determine the energy benefits achievable through the exploitation of timing speculation

While this chapter focuses primarily on how to optimize microarchitecture to improve energy efficiency at a non-zero error rate, other works have presented system architecture frameworks for exploiting application-level error resilience, which might enable a processor to operate at a non-zero error rate. ERSA [30] is an asymmetric multi-core architecture that contains cores with varying degrees of reliability. Each core belongs to one of two classes – super or strict-reliability cores (SRC) or relaxed-reliability cores (RRC). Typically, a single SRC is responsible for maintaining acceptable behavior for the processor. The SRC executes control-intensive code that is not tolerant to errors, schedules tasks on the RRCs, and checks for errors resulting from illegal memory accesses and timeouts of the RRCs

The RRCs are designed for reduced reliability targets, perhaps employing some of the techniques discussed in the previous section. These cores are the main source of throughput for an ERSA multi-core. Because of their relaxed correctness constraints, RRCs may be significantly less demanding of system or design resources than their counterpart SRCs. To enhance the feasibility of computing primarily on RRCs, software designed for ERSA should be built around error-resilient algorithms.

Work has been proposed to address the problem of task scheduling and

mapping on heterogeneous multi-core processors [81]. Such work is relevant to stochastic computing in that one axis of heterogeneity between cores may be reliability. The authors of [81] seek to achieve performance and energy efficiency by assessing the differences between tasks and between cores and finding the task-to-core mapping that optimizes a given efficiency metric.

Using a graph-based program representation called system-level instruction set architecture (SISA), [81] proposes to facilitate the task of mapping tasks to heterogeneous cores that typically have different ISAs. SISA represents programs as graphs with application characteristics such as data communication, length of computational tasks, reliability requirements, and task dependency. This representation allows for pre-running of tasks to identify the most efficient core on which to execute and dynamic task management to improve scheduling efficiency. SISA performs static scheduling based on integer linear programming.

5.6 Summary

The energy inefficiencies of traditional, conservative design approaches have led to the introduction of error-resilient design techniques that relax correctness in order to save power and energy. Until now, these design techniques have been applied to architectures that have been optimized for correctness.

This dissertation demonstrates that the energy-optimal error-free architecture may not be the optimal architecture for exploiting timing error resilience. In other words, one would make different, sometimes counterintuitive, architectural design choices when optimizing a processor to exploit timing speculation than when optimizing for correct operation. Consequently, the desired error rate and the error resilience mechanism should be taken into account when choosing the architecture for a timing speculative design. In addition to characterizing the effects of architectural optimizations on the slack and activity distributions, we have demonstrated that the optimizations can change the error rate behavior. Furthermore, we have demonstrated with experimental results for several DSP filter and general purpose architectures that optimizing architecture to exploit timing error resilience can significantly increase the energy efficiency of timing speculation. Energy efficiency benefits of up to 29% are achieved for Razor-based timing speculation.

CHAPTER 6

BINARY OPTIMIZATION FOR PROGRAMMABLE STOCHASTIC PROCESSORS

In this chapter, we advocate that binaries for timing speculative processors should be optimized differently than those for conventional processors to maximize the energy benefits of timing speculation. Since the program binary determines the utilization pattern of the processor, which in turn influences the error rate of the processor and the energy efficiency of timing speculation, binary optimizations for timing speculative processors should attempt to manipulate the utilization of different microarchitectural units based on their likelihood of causing errors. An exploration of targeted and standard compiler optimizations demonstrates that significant energy benefits are possible from timing speculation-aware binary optimization.

6.1 Introduction

Previous evaluations of the energy efficiency benefits of timing speculation have been based either on code compiled for a traditional target [19] – a processor that produces no errors – or code that relies on instruction set extensions and additional hardware support [82]. For example, [82] advocates the use of instruction set extensions whose circuit implementations have shorter critical paths. Unfortunately, physical design tools render most pipeline stages critical in power-optimized processors [27, 32], reducing the effectiveness of such approaches. Also, instruction set extensions may not be feasible in many settings.

In this chapter, we make a case for compiling differently for timing speculative processors in a way that increases energy efficiency without additional hardware support or instruction set extensions. To motivate our approach, we first reiterate the nature of benefits afforded by timing speculation (TS). The magnitude of energy efficiency benefits available from exploiting TS de-

pends on two factors – (a) *where* and (b) *how often* the processor produces errors when operating at an overscaled voltage or frequency. (For more details, see Section 4.1.1.) The path slack distribution of a timing speculative processor determines which paths do not meet timing constraints (negative slack paths) and thus cause errors when they are toggled. Likewise, the activity distribution of the processor describes how often paths are toggled, and thus determines the frequency of errors caused by a path when it has negative slack. Together, the slack and activity distributions dictate the error distribution of a processor, i.e., the locations and frequencies of errors produced in an overscaled processor.

Previous chapters have demonstrated that modifying the slack distribution (*where* errors are produced) can increase the energy efficiency of a timing speculative design [33, 48, 63, 64, 83]. We make a case for *timing speculation-aware binary optimization* by showing that even when the processor design and architecture are fixed (even if they are already optimized for a non-zero error rate), compiler optimizations can be used to modify the activity distribution (*how often* errors are produced) of a processor to enable more energy reduction for a non-zero error rate. Since the program binary, in conjunction with the processor architecture, determines a processor's activity distribution (which paths will be exercised and how often they will be exercised), optimizing a program binary can change the error rate behavior of the processor to improve the energy reduction afforded by timing speculation. For example, binary optimizations can be used to change the set of frequently exercised paths in a processor to avoid activating the longest paths. Since these paths are the first to have negative slack when the processor is overscaled, throttling their activity reduces early onset timing violations, enabling more overscaling and, consequently, lower energy for a given error rate. Similarly, binary optimizations can be used to reduce error rate by throttling activity in structures of the processor that cause the most errors. Other possibilities include optimizations to overlap errors in a single cycle to reduce the effective errors per cycle and optimizations to redistribute errors in the processors to reduce the effective error recovery overhead. For the case of voltage overscaling, all these optimizations have the effect of reducing the error rate for a given voltage, enabling the processor to operate at a lower voltage for a given error rate.

This chapter on timing speculation-aware binary optimization makes the

following contributions.

- We show that the activity distribution of a processor, and, by extension, the error distribution, can be altered through binary optimizations.
- We demonstrate that the energy efficiency of timing speculative processors can be improved by altering their activity distributions through binary optimizations without any additional hardware support.
- Through careful analysis of the main factors that influence processor error rate, we show that several optimizations that are already supported by existing compilers can improve the energy efficiency of TS.
- We quantify the energy savings from targeted and standard binary optimizations for a family of timing speculative processor architectures. We observe up to 39% additional energy savings from TS-aware binary optimization for a Razor-based processor.

The rest of this chapter is organized as follows. Section 6.2 describes the family of processor architectures that we use for developing specific compiler optimizations. Section 6.3 describes our experimental methodology. Section 6.4 discusses specific compiler optimizations and quantifies energy benefits provided by TS-aware compilation. Section 6.5 discusses related work. Section 6.6 summarizes the chapter.

6.2 Baseline Architecture

Which optimizations are most effective for a processor depend on which processor modules cause the most errors. In this section, we describe the family of processor architectures we study to develop binary optimization strategies and identify their error-critical modules. We also discuss how the error criticality of modules may depend on program characteristics.

6.2.1 FabScalar Architecture

We use the FabScalar [73] framework for our architectural evaluations. FabScalar is a parameterizable, synthesizable processor specification that allows

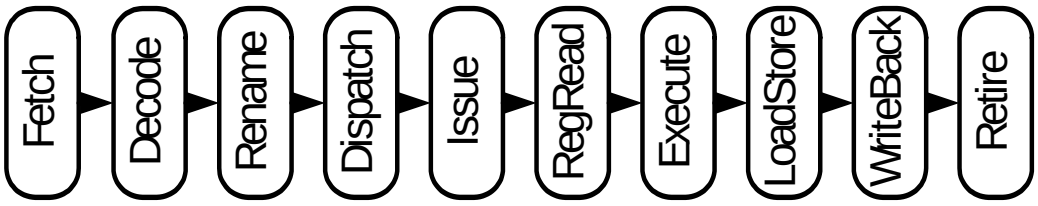


Figure 6.1: The FabScalar Pipeline. [73]

for the generation and simulation of RTL descriptions for arbitrarily configured scalar and superscalar processor architectures. FabScalar allows for the configuration of many microarchitectural parameters, including superscalar width (ss), fetch width and depth (fw, fd), numbers and types of functional units, issue width and depth (iw, id), issue queue size (iq), select logic depth (sel), register file depth (rrd), re-order buffer entries (rob), physical registers (reg), and load and store queue sizes (lsq). In this chapter, we study a family of superscalar processors by selecting interesting candidates from the available configurations space of FabScalar. Figure 6.1 shows the FabScalar pipeline.

6.2.2 Error Criticality Analysis

Different pipeline stages cause errors at different rates, depending on their slack and activity distributions. Figure 6.2 shows the static slack distributions for the pipeline stages that cause the most errors. While our highly optimized design flow removes excess slack in all stages, two stages in particular – the issue queue (IQ) and the load-store unit (LSU) – have the highest number of critical paths. Based on Figure 6.2, one might expect that the IQ, having many more critical paths than all other modules combined, would produce the most errors in the processor. However, the static slack distribution only shows the *potential* for paths to cause errors. Not all stages exercise their critical paths often. Stages with frequently exercised critical paths cause the most errors. In Figure 6.3, we create activity-weighted, *dynamic* slack distributions by showing the sum of toggle rates for all the paths at each value of timing slack (activity from SPEC benchmarks). The more timing critical *activity* a module has, the more errors it is likely to cause. From Figure 6.3, it is clear that the LSU dominates the error distribution of the processor.

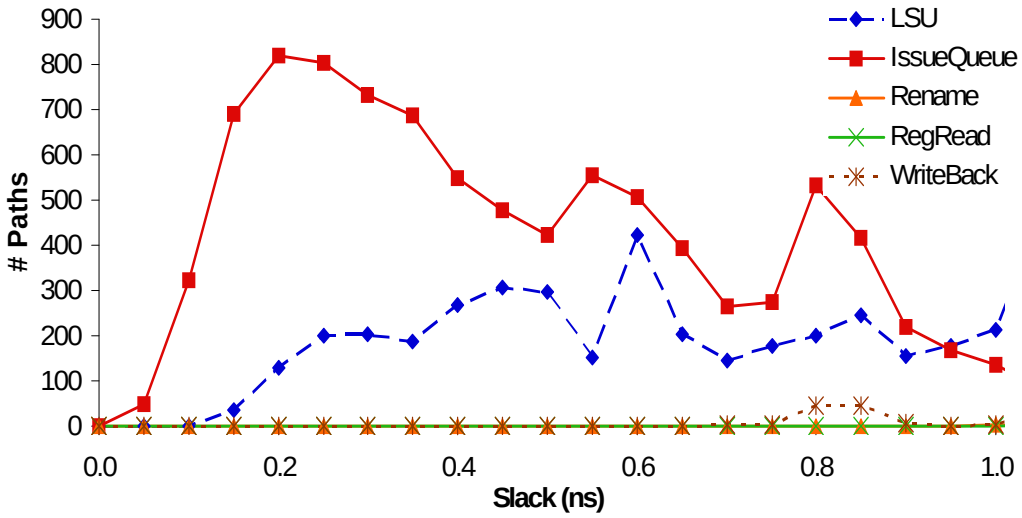


Figure 6.2: The static slack distributions for the pipeline stages show how many critical paths they have but do not provide information about how often the paths toggle, which is essential in characterizing error rate.

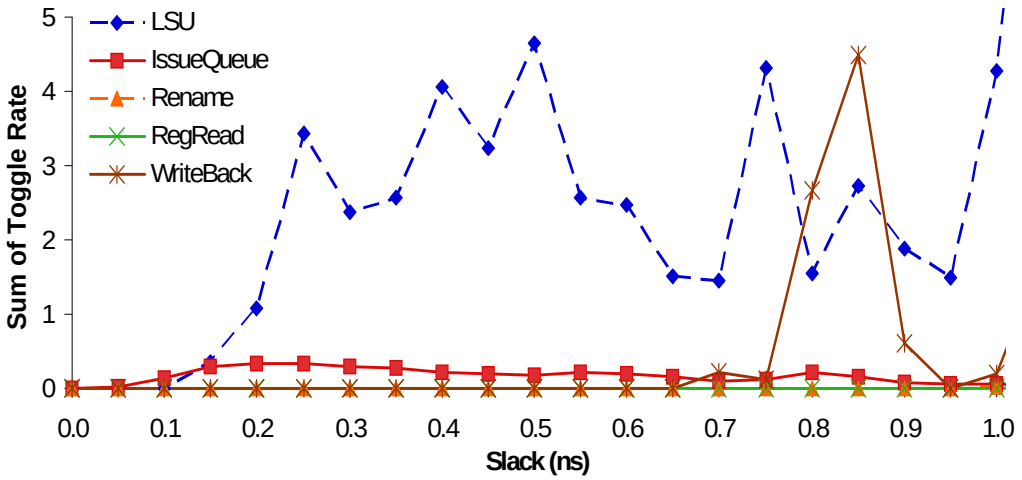


Figure 6.3: The activity-weighted (dynamic) slack distributions for different pipeline stages indicate how much timing critical activity they exhibit and, by extension, how frequently they will produce errors for a given level of overscaling – i.e., a given (voltage, frequency) pair.

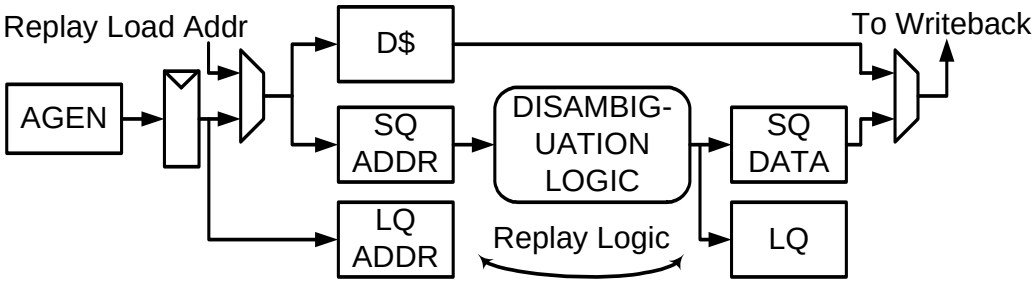


Figure 6.4: Memory disambiguation is on the critical path of the LSU [73]. The path delay is longest when store-to-load forwarding is required, since this necessitates an access to the SQ data RAM, in addition to the other disambiguation operations.

6.2.3 Program Dependence of Error Criticality

As demonstrated in the previous section, the LSU and, secondarily, the IQ are the primary sources of timing violations for the family of processor architectures that we studied. Below, we describe the implementation of the LSU and the IQ in the FabScalar processor to understand the dependence of error rate on program characteristics.

The LSU (Figure 6.4) performs memory disambiguation for the processor. This involves checking for dependencies between loads and stores. After address resolution, a store must search the address CAM of the LQ and process all entries with matching addresses to determine if any load issued out of order and broke a RAW dependence. Load disambiguation is more complicated because it may include store-to-load forwarding. In addition to a search through the SQ address CAM, a load must generate a mask vector indicating all preceding stores in program order. Matching entries from the CAM search are filtered by the mask vector, and the latest resulting entry, if any, forwards data from the SQ data RAM to the load.

LSU delay depends on program characteristics for several reasons. The primary reason is that the store-to-load forwarding path is on the static critical path of the LSU. Since many RAW dependencies in a code lead to more forwarding, the timing error rate will be higher for code with a relatively large number of RAW dependencies. Program characteristics also determine the utilization of the LQ and the SQ, which, in turn, dictates access delays for the structures. For example, when the LQ or SQ are nearly full, as may be common for memory-centric codes, more entries must be accessed in a single

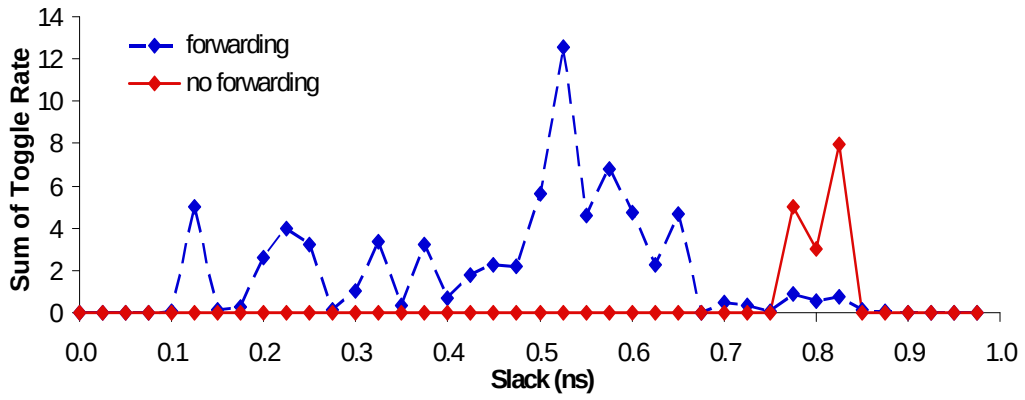


Figure 6.5: Since forwarding paths are critical in the LSU, eliminating dependencies and the need for store-to-load forwarding reduces activity on the critical paths of the LSU.

cycle to generate mask vectors, increasing the length of the propagation path and, consequently, increasing delay. Additionally, when there are many dependencies between memory operations, address CAM searches generate many hits, increasing load capacitance and delay for the CAM access. Finally, propagation delay increases when many hits are signaled in parallel (due to many potential dependencies), since the average length of the propagation path from the CAM entries to the port increases. Hence, the average delay is higher for memory-centric codes with a large number of dependencies.

We confirmed that the forwarding paths are timing critical in the LSU, and that more dependencies result in activation of longer paths, by observing the activity-weighted (dynamic) slack distribution of the LSU for two different instruction streams (Figure 6.5). The first contains a stream of memory operations that access the same address. Each load depends on the previous store and activates the forwarding paths in the LSU. In the second stream, the dependencies are removed. Figure 6.5 demonstrates that activity on the critical paths of the LSU is greatly reduced when the dependencies are removed and forwarding is not required.

The wakeup-select logic used in the IQ is similar in nature to the memory disambiguation logic in the LSU. For example, wakeup consists of finding all instructions that depend on the destination register of another instruction. This CAM-based dependence check in the IQ is performed in much the same way as the dependence checks in the LSQ. Likewise, select logic, which selects

a ready, waiting instruction to execute is somewhat akin to the masking logic that identifies valid, conflicting stores for forwarding. Because of their similarities, the LSU and IQ have similar timing considerations.

6.3 Methodology

To understand the impact of different binary optimizations on the error behavior and energy efficiency of different processor architectures, we used a detailed methodology that carefully models the relationships between execution behavior, power, performance, and reliability. Designs are implemented with the TSMC 65GP library (65 nm), using Synopsys Design Compiler for synthesis and Cadence SoC Encounter for layout. In order to evaluate the power and performance of designs at different voltages and to provide V_{th} sizing options for synthesis, Cadence Library Characterizer was used to generate low, nominal, and high V_{th} libraries at each voltage (V_{dd}) between 1.0 V and 0.5 V at 0.01 V intervals. Designs are implemented at 500 MHz. Power, area, and timing analyses are performed in Synopsys PrimeTime. Gate-level simulation is performed with Cadence NC-Verilog to gather activity information for the design, which is subsequently used for dynamic power estimation and error rate measurement. Please refer to Section 4.2.4 for a detailed description of error rate measurement.

In addition to inducing timing errors by increasing logic delays, voltage scaling may prompt reliability concerns for SRAM structures, such as insufficient static noise margin (SNM). Fortunately, the minimum energy voltage for our processors is around 750 mV, while production-grade SRAMs have been reported to operate reliably at voltages as low as 700 mV [79]. Research prototypes have been reported to work for even lower voltages. In any case, modern processors typically employ a "split rail" design approach, with SRAMs operating at the lowest safe voltage for a given frequency [58].

In our evaluations, we compile and run several microbenchmarks to demonstrate architecture-specific TS-aware optimizations. We also run instruction traces from the SPEC benchmark suite (bzip, gap, mcf, parser, vortex), after fast-forwarding the benchmarks to their Simpoints [59]. All benchmarks are compiled with gcc-2.7.2.3 (SPEC benchmarks and gcc version correspond to those supported by FabScalar).

Table 6.1: Average processor-wide Razor overheads.

Hold buffering	Razor FF	Counterflow	Error recovery
2% energy	23% energy	<1% energy	P cycles

We model Razor-based error resilience [19] in this chapter (though our proposed techniques are generally applicable to any timing error-resilient processor). Table 6.1 summarizes the processor-wide static and dynamic overheads of Razor-based error detection and correction. In our design flow, we measure the percentage of die area devoted to sequential elements, as well as the timing slack (with respect to the shadow latch clock skew of 1/2 cycle) of any short paths that need hold buffering. When evaluating energy at the processor level, we account for the increased area and power of Razor flip-flops, hold buffering on short paths, and implementation of the recovery mechanism. Most of the static overhead is due to Razor FFs. Buffering overhead is small, and the availability of cells with high and low V_{th} provides more control over path delay, eliminating the need for buffering on most paths. We also add energy and throughput overheads proportional to the error rate to account for the dynamic cost of correcting errors over multiple cycles. We use a counterflow pipeline Razor implementation [19] with correction overhead proportional to the number of processor pipeline stages (P). We conservatively replace all sequential cells with Razor FFs.

6.4 Experimental Results

We now discuss different architecture-specific binary optimizations that may increase the efficiency of timing speculative processors. The proposed optimizations are primarily geared toward error avoidance in the LSU and IQ. We first discuss targeted loop-based optimizations and quantify their benefits through the use of microbenchmarks. Then, we evaluate the benefits of combining standard gcc optimizations using O levels for SPEC benchmarks.

6.4.1 Targeted Optimizations for TS Processors

Loop Unrolling: As described above (Section 6.2), activity on the static critical paths of the LSU can be reduced by avoiding dependent memory

operations and scenarios that cause the LSQ to fill up. This can enable significantly deeper voltage overscaling, since the LSU is often the source of many timing violations.

Loop unrolling is a classic compiler optimization that can eliminate and spread out loop carried dependencies, and thus has the potential to reduce LSU delay. Normally, unrolling would only be used when spin up and spin down costs are overcome by reducing the number of executed instructions. However, TS-aware compilation provides a new use for unrolling – avoiding errors to increase the efficiency of TS by grouping often independent instructions (like vector math) and eliminating often dependent instructions (like branches and loop index updates). Unrolling also allows optimization of register allocation over multiple loop iterations that can eliminate load and store disambiguation, thus reducing pressure on the LSU. Unrolling can also reduce pressure on the branch resolution unit and arithmetic unit, since the number and frequency of branch instructions and loop index updates are reduced. Thus, in addition to fostering critical path avoidance by reducing dependencies, loop unrolling can also be an agent for activity throttling.

Unrolling can cause binary size to increase, which may reduce instruction cache efficiency and may be undesirable in some embedded processors. Unrolling may also cause an increase in dynamic power. When exploiting TS-aware binary optimization, it is important to consider the impact on performance and power, as well as energy efficiency.

Figure 6.6 shows an example of loop unrolling by a factor of four. Figure 6.7 shows the error rate of the processor when executing the two code sequences of Figure 6.6. Unrolling significantly reduces the error rate by reducing activity on the forwarding paths in the LSU. This error rate reduction enables additional overscaling and results in a substantial energy reduction for a Razor-based TS processor, as shown in Table 6.2. Microarchitectural parameters not specified in Table 6.2 are iq=16, rob=64, reg=64, lsq=8+8=16.

In the error-free case, the same unrolled loop causes dynamic power to increase significantly, even as it increases throughput. Thus, unrolling has the potential to reduce error rate but may also increase power for a conventional processor where TS is not allowed. So, most energy-efficient binary optimization depends on whether the target uses TS. This demonstrates the need for TS-aware compiler analysis and optimization.

```
for(i=0; i<N; i++)
  sum += A[i];
```

```
for(i=0; i<N; i+=4){
  sum1 += A[i];
  sum2 += A[i+1];
  sum3 += A[i+2];
  sum4 += A[i+3];
}
sum=sum1+sum2+sum3+sum4;
```

Figure 6.6: Original loop (left) and unrolled loop (right).

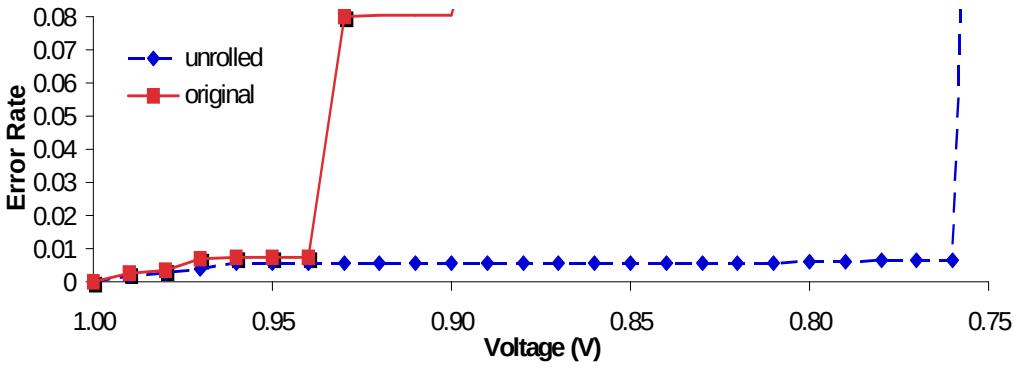


Figure 6.7: Loop unrolling reduces activity on LSU forwarding paths, resulting in a significant error rate reduction.

Table 6.2: Razor-based TS and error-free energy savings (%) for loop unrolling. (ss = superscalar width)

CORE	original	unrolled	unrolled error-free
$ss1$	11.8	43.1	1.6
$ss2$	6.4	20.8	2.0
$ss4$	4.0	42.9	3.2

Balancing Instruction-Level Parallelism: In an out-of-order processor, instructions are dispatched to the processor backend as long as there is available space in the appropriate backend structures, namely, the reorder buffer (ROB), IQ, and LSQ. However, when there are not enough execution units to handle ready, waiting instructions, backend structures fill up and remain full. As discussed above, this leads to longer propagation delays for these structures – especially for queues.

Thus, we observe that when hardware parallelism is limited, optimizing the binary to promote software parallelism can actually increase energy in a timing speculative processor by increasing logic delay and limiting overscaling. Consequently, when hardware parallelism is limited, a TS-aware compiler should actually throttle parallelism to prevent instructions from reaching the backend. This kind of compiler optimization is contrary to conventional wisdom, which promotes ILP whenever possible for potential performance gains.

On the other hand, when hardware parallelism is available, the scenario is reversed. Dependencies that hinder ILP keep queues full and increase the delay of dependence-checking logic. Thus, when adequate hardware resources are available, enhancing parallelism can eliminate dependencies and lead to better TS efficiency.

To illustrate the above points, we have run the codes in Figure 6.8 on TS processors with different superscalar widths. Figure 6.9 compares the error rates of the code sequences for the $ss1$ case. In this case, hardware parallelism is not available, and exposing more instructions to the processor backend causes queue structures to fill, increasing propagation delays. Thus, the error rate increases as more parallelism is exposed (e.g., ILP4).

For a processor with more hardware parallelism (e.g., $ss2$), the backend can handle increased software parallelism without putting excessive fill pressure on queue structures. In this case, the reduced dependencies of the more parallel code reduce activity in the timing critical disambiguation logic and enable more overscaling. Figure 6.10 compares error rates for the codes on a $ss2$ processor. The error rate for the code without exposed parallelism (ILP1) increases abruptly and surpasses the error rates for the more parallel codes. Table 6.3 shows energy results for Razor-based TS, demonstrating that TS efficiency increases *when hardware and software parallelism are balanced*. The table also demonstrates that enhancing parallelism does not provide any

```
for(i=0; i<N; i++)
  sum += A[i];
```

```
for(i=0,j=N/2;i<N/2;i++,j++){
  sum1 += A[i];
  sum2 += A[j];
}
sum = sum1+sum2;
```

Figure 6.8: Original code (left – ILP 1) and code with more ILP exposed (right – ILP 2).

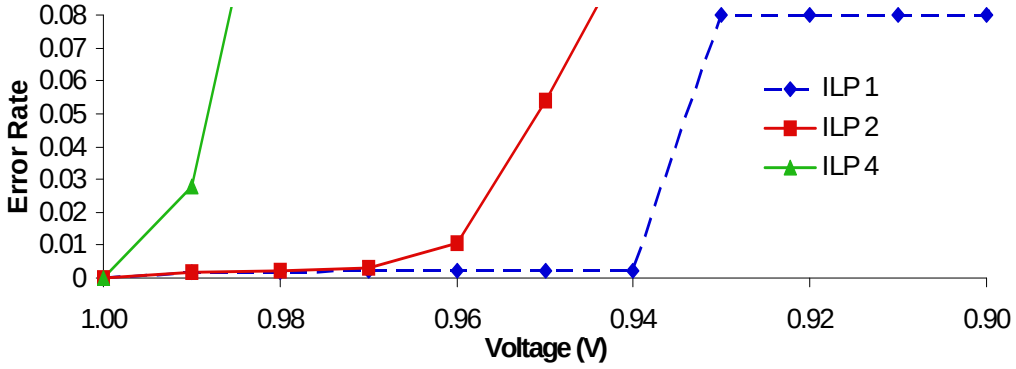


Figure 6.9: When hardware parallelism is not available ($ss1$), exposing parallelism floods backend queue structures and increases the error rate.

significant energy savings in the error-free case, motivating the need for TS-specific compiler analysis and optimization.

Loop Splitting: Loop splitting or peeling can also be used to break dependencies in code by peeling dependent instructions out of the loop body. The original code in Figure 6.11 contains two dependencies – a loop carried dependence for the accumulator variable (sum), and a dependence between the array indices (i, j). By peeling one of the iterations from the loop, we can eliminate one of the dependencies. This reduces the load on the CAM structure that performs dependence checking, and eliminates occurrences of forwarding. Figure 6.12 shows how peeling a dependence from the loop re-

Table 6.3: Razor-based TS and error-free energy savings (%) for balancing parallelism.

CORE	ILP 1	ILP 2	ILP 4	ILP 2 error-free
$ss1$	13.1	5.5	0.0	1.4
$ss2$	5.4	9.8	9.7	0.8

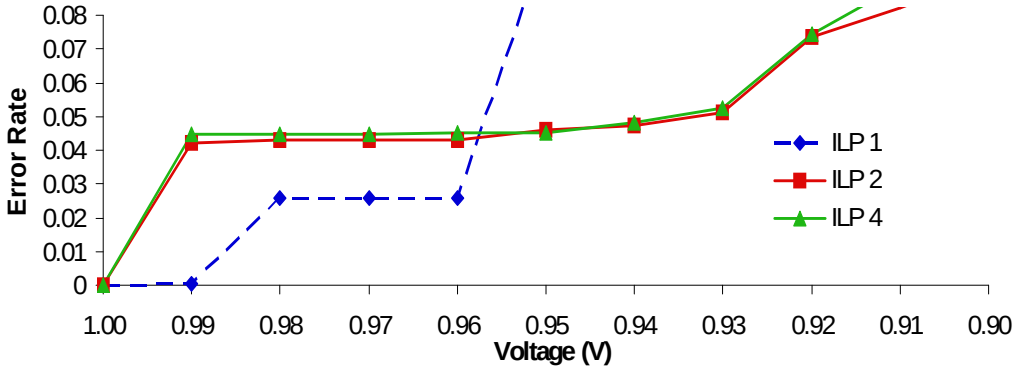


Figure 6.10: When hardware parallelism is available ($ss2$), exposing parallelism eliminates dependencies and reduces error rate.

```
j = N-1;
for(i=0; i<N; i++){
  sum += A[i] + A[j];
  j = i;
}
```

```
sum = A[0] + A[N-1];
for(i = 1; i < N; i++){
  sum += A[i] + A[i-1];
}
```

Figure 6.11: Original code (left) and code with a dependence peeled from the loop (right).

duces the error rate for $ss1$, $ss2$, and $ss4$ processors. Table 6.4 compares the energy savings achieved by Razor-based TS and error-free operation before and after loop splitting is performed. In all cases, the additional overscaling enabled by loop splitting results in energy savings for Razor-based TS. For error-free operation, loop splitting actually increases energy slightly, because it causes a small reduction in performance (IPC). This divergence between the best decision for TS and error-free cases motivates the need for TS-specific compiler analysis and optimization.

Loop Fusion: Another technique for manipulating dependence patterns in

Table 6.4: Razor-based TS and error-free energy savings (%) for loop splitting.

CORE	original	split	split error-free
$ss1$	5.8	13.8	-0.2
$ss2$	0.0	9.0	-0.4
$ss4$	3.6	13.4	-0.1

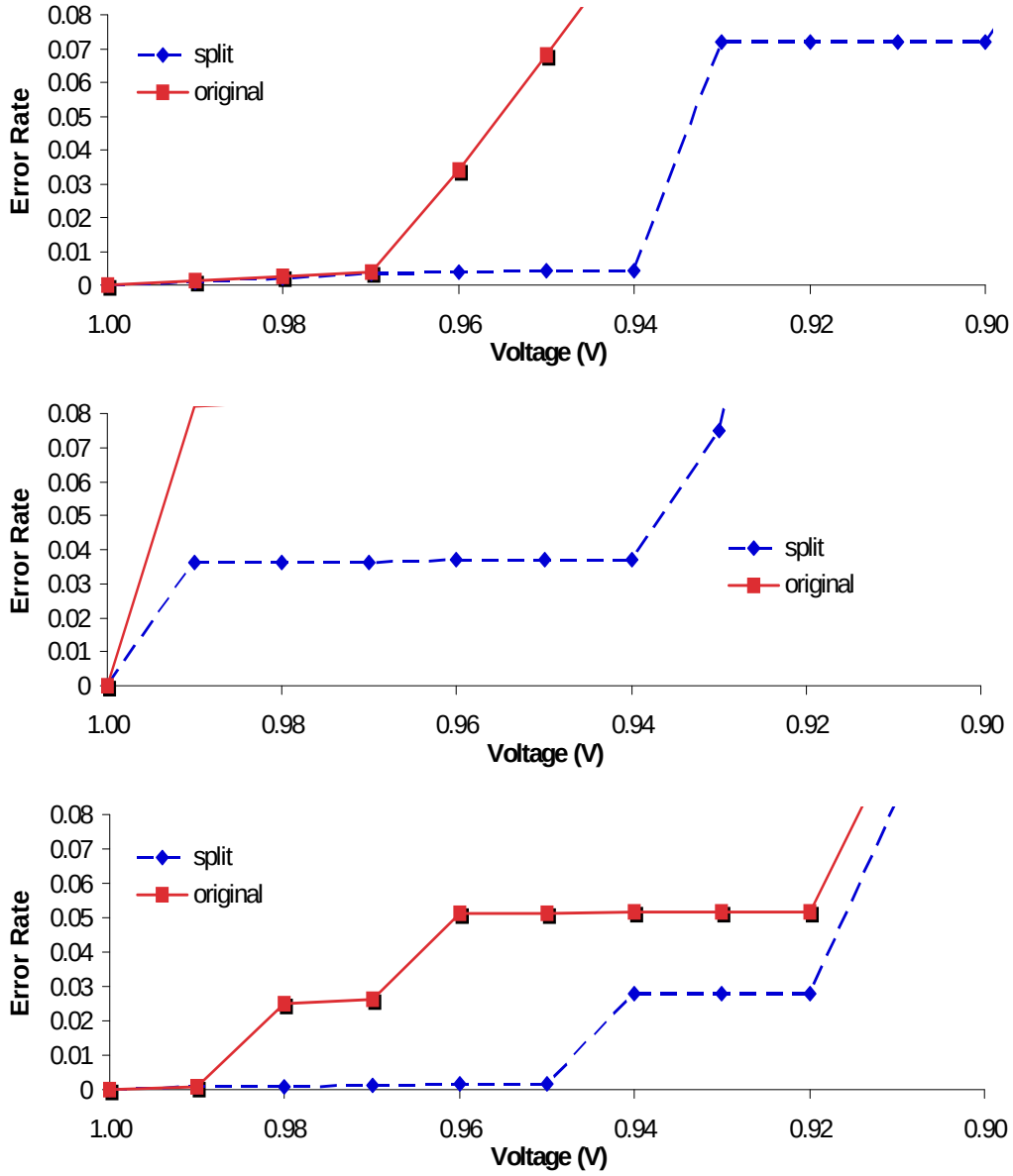


Figure 6.12: By removing a dependence from the loop, loop splitting reduces the error rates of the $ss1$ (top), $ss2$ (middle), and $ss4$ (bottom) processors.

```
for(i=0; i<N; i++)
  sum1 += A[i];
for(i=0; i<N; i++)
  sum2 += B[i];
for(i=0; i<N; i++)
  sum3 += C[i];
for(i=0; i<N; i++)
  sum4 += D[i];
```

```
for(i = 0; i < N; i++){
  sum1 += A[i];
  sum2 += B[i];
  sum3 += C[i];
  sum4 += D[i];
}
```

Figure 6.13: Original code (left) and code with fused loops (right).

code is loop fusion. Loop fusion merges independent instructions in separate loops into the same loop. Grouping independent instructions can help to break up long chains of dependent instructions by spreading them farther apart in the binary. This can reduce the need for forwarding, since conflicting instructions are able to clear the LSQ before their dependent instructions are dispatched to the processor backend. As a side effect, loop fusion may decrease locality of access, which can degrade cache performance. In general, it is important to consider the potential performance impacts of TS-aware binary optimization along with the energy savings it enables.

Figure 6.13 compares code sequences with (right) and without (left) loop fusion. Note that loop fusion and loop splitting are inverse operations; that is, the original code can be produced by performing loop splitting on the fused code. In the $ss1$ case (Figure 6.14), grouping independent instructions does not provide benefits, since there are not adequate hardware resources to handle the exposed ILP. In this case, the unfused (split) code has a lower error rate, because the activity of the LSU (the module that causes the most errors) is throttled by the interleaving of branches and loop index updates with the loads and stores. This activity throttling leads to increased TS energy efficiency, as shown in Table 6.5.

In the $ss4$ case (Figure 6.15). the clustering of independent instructions in the fused code spaces out dependent instructions in the pipeline, thus eliminating many occurrences of forwarding and reducing activity on timing critical paths in the LSU. This critical path avoidance reduces error rate and enhances TS efficiency, as shown in Table 6.5. Again, energy savings from loop fusion in the error-free case are only meager (<1%), motivating the need for TS-aware compiler analysis and optimization.

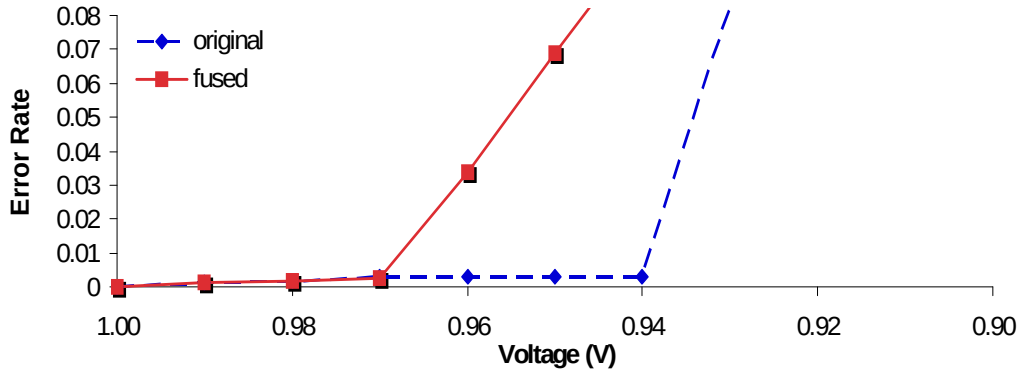


Figure 6.14: When hardware parallelism is limited ($ss1$), the unfused (split) code has a lower error rate, since LSU activity is throttled.

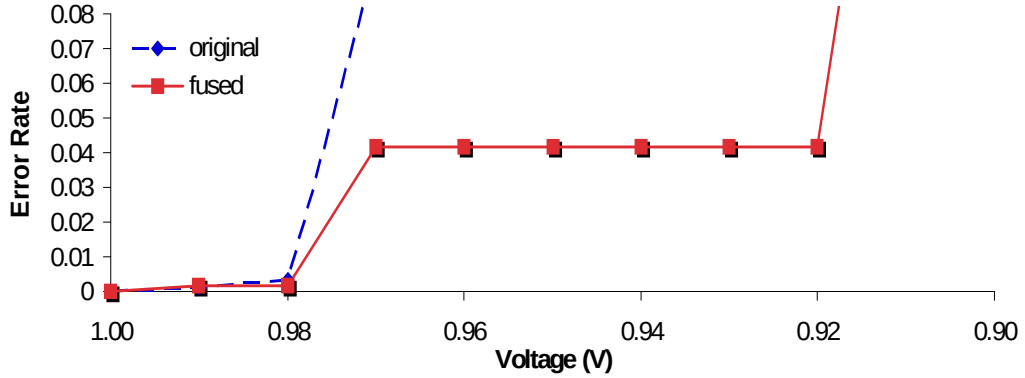


Figure 6.15: When hardware parallelism is available ($ss4$), the fused code spaces out dependent instructions, reducing forwarding and, consequently, error rate.

Table 6.5: Razor-based TS and error-free energy savings (%) for loop fusion.

CORE	original	fused	fused error-free
$ss1$	12.2	5.9	0.2
$ss4$	4.2	12.3	0.5

Several other TS-aware binary optimizations are possible. The goal of this chapter is to demonstrate that significant energy benefits may be possible from TS-aware binary optimization. An exhaustive exploration of all possible binary optimizations is beyond the scope of this work.

6.4.2 Standard gcc Optimizations for Timing Speculative Processors

Fortunately, many standard gcc optimizations have goals similar to the targeted optimizations discussed above. For example, optimizing for a higher O level has the potential to reduce dependencies and bolster ILP. Similarly, optimizing for a lower O level may effectively restrict ILP. Below, we evaluate the TS efficiency of SPEC binaries that have been optimized at different O levels.

For architectures without available hardware parallelism (e.g., $ss1$), highly optimizing compute-limited applications can cause pipeline backend structures to fill, resulting in longer delays and higher error rates. On the other hand, for memory-bound applications with many indirect memory references, critical LSU paths are not frequently exercised. Instead, IQ contributes most substantially to the error rate, so optimizing at a higher O level, which reduces average IQ entries and, consequently, IQ delay, reduces the error rate. Thus, when hardware parallelism is limited, compute-limited applications should be optimized for a lower O level ($O0$), while memory-bound, pointer-chasing codes can be optimized for a higher O level.

For architectures with available hardware parallelism (e.g., $ss2$), highly optimizing compute-limited applications can reduce dependencies, activity on critical LSU paths, and error rate. Optimizations do not have much effect on memory-bound, pointer-chasing codes, since available hardware parallelism allows average IQ entries to remain low, and critical LSU paths are not frequently exercised. Below, we test these intuitions for SPEC benchmarks with standard gcc O levels.

Figure 6.16 shows the error rates of SPEC benchmarks we evaluated at available O levels, running on the $ss1$ core. Although higher optimizations (e.g., $O2$) generally improve performance (IPC), they increase error rate and degrade TS energy efficiency for compute-limited codes (Table 6.6). This is

because optimizing at the higher O level enhances software parallelism, but there is not sufficient hardware parallelism to handle the dispatched instructions. Thus, backend structures (LSQ and IQ) fill and propagation delay increases, limiting overscaling. Consequently, performing no optimizations ($O0$) is preferable for compute-limited applications on the $ss1$ core when TS is used. Note that this is an interesting result, as the choice of O level would be different when compiling for the error-free case, since increasing the O level improves performance.

For pointer-chasing codes like vortex, which performs object-oriented database lookups, and thus contains many indirect memory references, critical LSU forwarding paths are not frequently exercised. Rather than the LSU, the IQ dominates the processor error rate for the $O0$ binary on this core. Optimizing for a higher O level results in fewer average IQ entries, reducing delay and error rate, and significantly increasing energy savings (Table 6.6).

For the $ss2$ core, the backend queue structures are not overly stressed. Optimizing at a higher O level reduces dependencies for compute-limited codes and, by extension, activity on the critical paths of the LSU. This reduction in critical path activity reduces error rate (Figure 6.17) and allows more overscaling and reduced energy (Table 6.7). Thus, higher optimization (O) levels are beneficial, in general, for Razor-based TS when hardware parallelism is not restricted. Choosing the correct optimization level that balances hardware and software parallelism maximizes energy savings. Note that results in this section demonstrate that the best optimization level is different for TS and non-TS cases. For example, $O1$ achieves the most energy benefits for TS on the $ss2$ core, even though $O2$ has higher performance in the error-free case.

As expected, memory-bound, pointer-chasing codes see little impact from optimizations on the $ss2$ core. The many indirect memory references in vortex cannot be optimized at compile time, and thus, optimizations do not significantly impact LSU activity. Also, since HW parallelism is available to relieve IQ fill pressure, optimizations do not significantly reduce the IQ error rate either.

In the error-free case, optimizing at a higher level ($O2$) can increase performance, but this performance comes with a significant increase in power consumption. Thus, energy is not significantly improved with $O2$ in the error-free case (Tables 6.6, 6.7). Distinctions between the best strategy in

Table 6.6: Razor-based TS and error-free energy savings (%E), performance (IPC), and binary size (MB) for SPEC benchmarks at different O levels ($ss1$).

$ss1$	bzip			mcf			vortex		
OPT	%E	IPC	MB	%E	IPC	MB	%E	IPC	MB
$O0$	11.8	0.45	0.32	14.7	0.56	0.31	0.0	0.55	1.70
$O1$	7.5	0.79	0.29	9.2	0.67	0.29	14.0	0.49	1.48
$O2$	0.0	0.77	0.29	9.0	0.54	0.29	14.0	0.51	1.47
$O3$	7.2	0.75	0.31	9.2	0.59	0.30	14.0	0.51	1.49
$O2$ no-error	1.2	0.77	0.29	0.1	0.56	0.29	0.0	0.55	1.47

Table 6.7: Razor-based TS and error-free energy savings (%E), performance (IPC), and binary size (MB) for SPEC benchmarks at different O levels ($ss2$).

$ss2$	bzip			mcf			vortex		
OPT	%E	IPC	MB	%E	IPC	MB	%E	IPC	MB
$O0$	0.0	0.65	0.32	0.0	0.69	0.31	10.4	0.61	1.70
$O1$	7.7	1.39	0.29	13.4	1.45	0.29	10.0	0.74	1.48
$O2$	5.7	1.32	0.29	9.1	1.37	0.29	10.2	0.75	1.47
$O3$	7.5	1.34	0.31	8.5	1.26	0.30	10.4	0.76	1.49
$O2$ no-error	1.2	1.5	0.29	1.0	1.49	0.29	0.4	0.78	1.48

TS and non-TS cases further demonstrates the need for TS-aware compiler analysis and optimization.

6.5 Related Work

Work on TS-aware design discussed in previous chapters has focused on optimizing *hardware* to improve the efficiency of TS. Work has been done primarily at the design level [32, 33, 48, 63, 64] and the architecture level [83] to reshape the slack distribution of a processor to enhance the energy efficiency benefits of TS. These optimizations primarily focus on making the static slack distribution of a processor more amenable to overscaling.

This work, however, focuses on optimizations at the *software* level that influence the activity and *dynamic* slack distributions of a processor (see Section 6.2). Since the error rate of a timing speculative processor depends on both slack and activity (see Section 4.1.1), TS-aware compilation has just

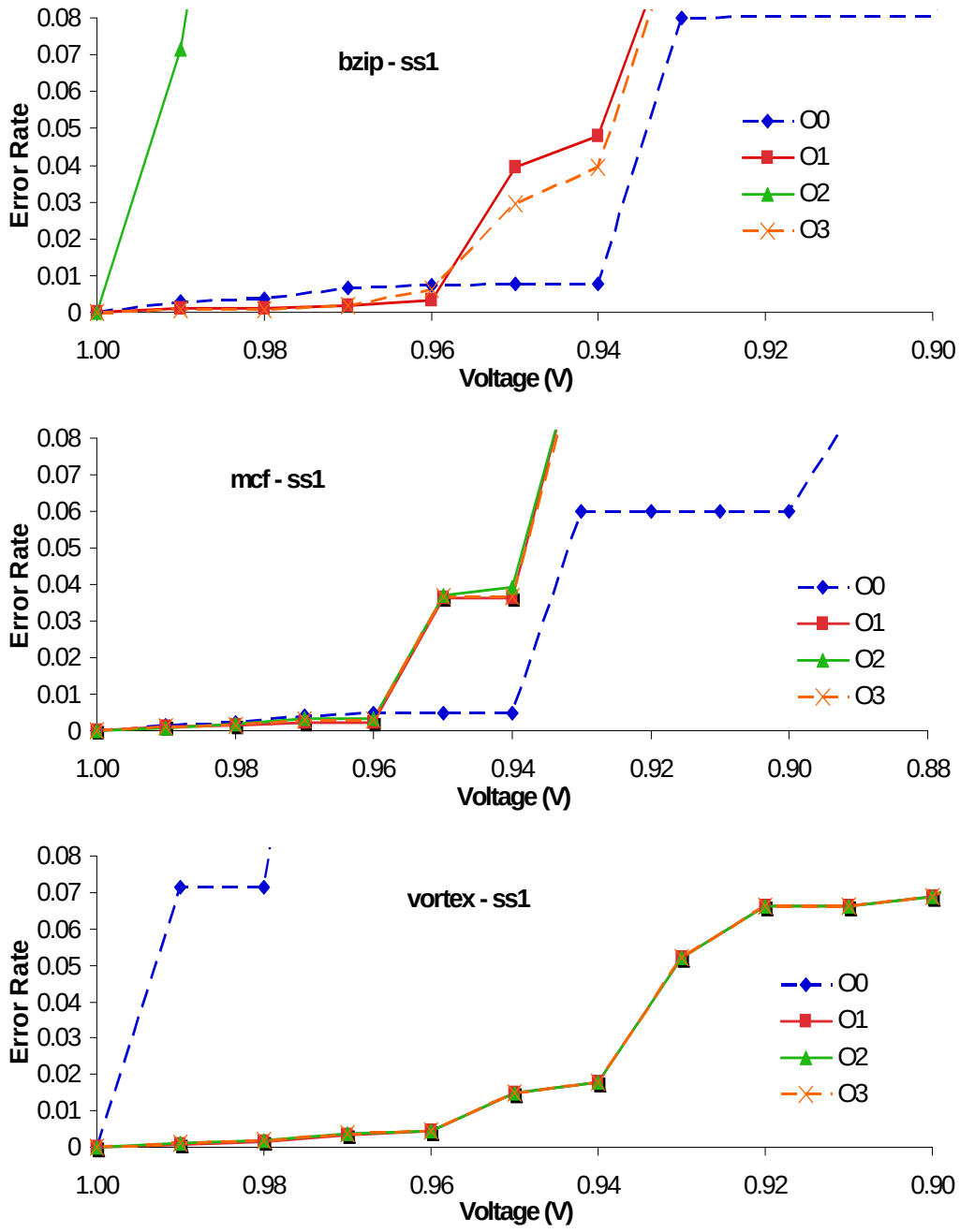


Figure 6.16: For the *ss*1 core, highly optimizing compute-bound code (e.g., bzip) can increase the error rate, because fill pressure increases the delays of highly utilized pipeline backend structures and limits overscaling. Optimizing memory-bound code (e.g., vortex) can reduce error rate, because critical LSU paths are not exercised, and optimizations reduce IQ fill pressure.

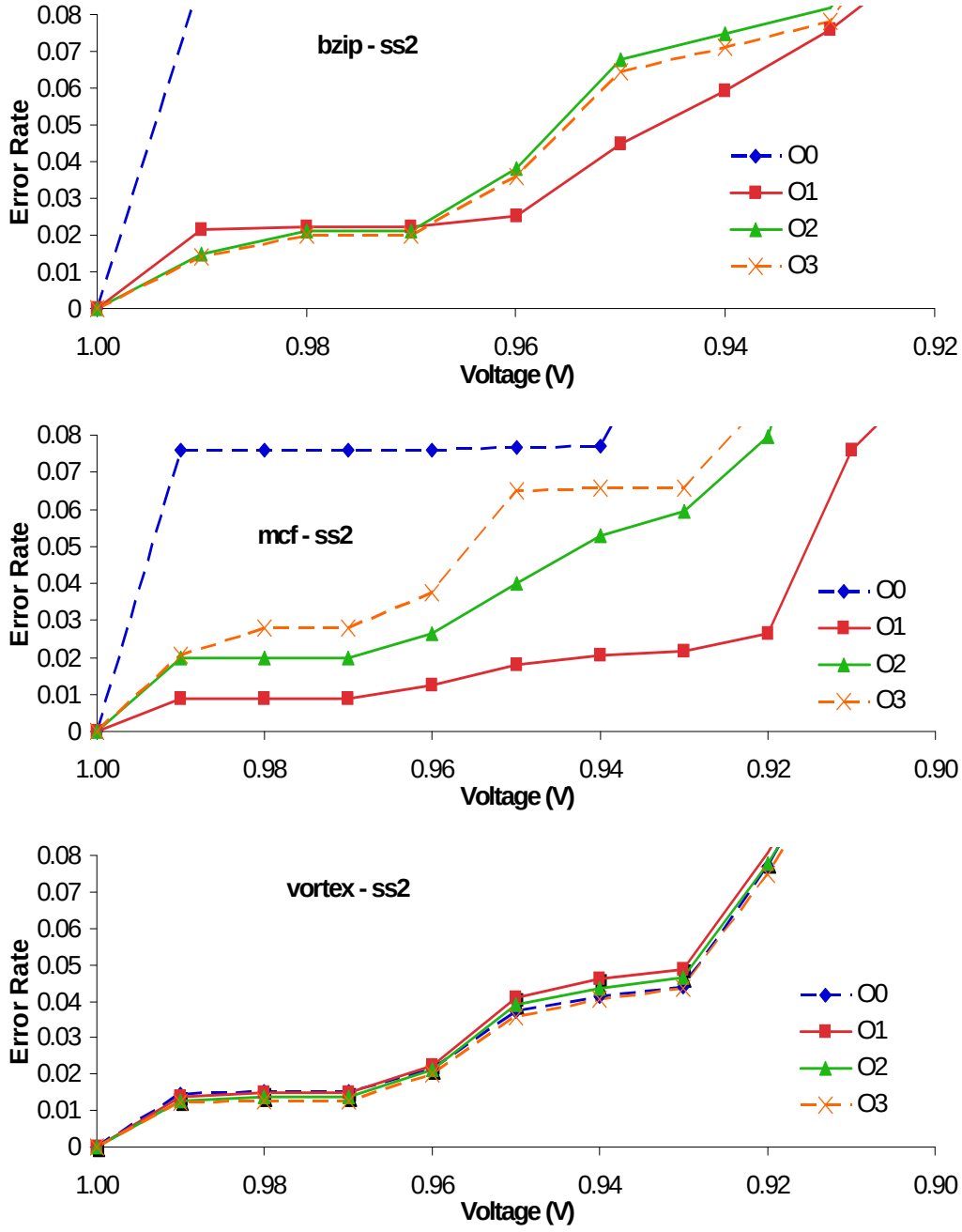


Figure 6.17: Optimizing compute-bound code (e.g., bzip) can reduce dependencies and activity on the critical paths of the LSU for the *ss2* core. Choosing the right optimization level that balances HW and SW parallelism can be an important factor in reducing processor error rate. The effect of optimizations is limited for memory-bound code (e.g., vortex).

as much potential to optimize processor error rate as do hardware-based techniques. A promising direction of work involves co-optimization of software and hardware to reshape the dynamic slack distribution and maximize the energy efficiency benefits of exploiting TS.

The closest related work [82] focuses on extending the instruction set to include instructions for which the circuit implementation has a shorter critical path. Replacing instructions with these new instructions increases timing slack and enables more overscaling. The instruction set extensions proposed by [82] primarily focus on reduced-complexity arithmetic operations. We optimize program binaries to improve energy efficiency for TS processors without requiring hardware support.

In a typical ASIC design flow, all paths with excess timing slack are optimized to remove the timing slack, thus reducing power consumption and area. This design style produces a design with a critical slack wall [27], so that the vast majority of timing paths have near-critical slack. Since all circuit modules in our designs have many critical paths, as we would expect in a processor implemented by a typical CAD flow, we are unable to utilize optimizations that redirect instructions to units with more timing slack [82]. Instead, our optimizations focus on avoiding activation of the critical paths in a hardware unit and throttling the activity of units that cause the most errors. Additionally, we focus on binary optimizations that do not require instruction set extensions and, thus, may be more generally applicable. Finally, since the architectures that we evaluate are different than the architecture studied in [82], the modules that cause the most errors are different. Therefore, our architecture-specific optimizations focus on different regions of the processor.

6.6 Summary

Previous work on improving energy efficiency of timing speculative processors relied on code targeting conventional processors or assumed additional hardware support and instruction set extensions. In this chapter, we have demonstrated that careful binary optimization can increase the energy efficiency of error-resilient processors without additional hardware support. Since the program binary determines the utilization pattern of the proces-

sor, which in turn influences the error rate of the processor and the energy efficiency of timing speculation, optimizing a binary specifically for timing speculative processors can manipulate the utilization of different microarchitectural units based on their timing slack distribution to deliver energy efficiency benefits. We have demonstrated up to 39% additional energy savings with timing speculation-aware binary optimization for Razor-based processors. We expect the energy benefits to grow as more sophisticated compiler techniques are developed.

CHAPTER 7

A PROGRAMMABLE STOCHASTIC PROCESSOR PROTOTYPE

While the previous chapters demonstrate significant benefits and potential for programmable stochastic processors, going forward, more work is needed to enhance the generality of stochastic computing techniques. In Chapter 3 we presented a first example of a programmable stochastic processor that improves energy efficiency by selecting between multiple functional units based on the desired operational error rate. In this chapter, we present a more elaborate evaluation of a programmable stochastic processor prototype that demonstrates energy (performance, runtime reduction) benefits for applications running on a commodity processor. Performance benefits are gleaned from careful relaxation of correctness that eliminates inefficiency and exposes errors in error-tolerant applications. We perform evaluations for data intensive applications that run on general purpose graphics processing units (GPGPUs). In order to improve performance for our target applications, we identify inefficiencies in the applications (control and memory divergence) and present techniques that eliminate these inefficiencies by carefully relaxing correctness for a subset of control and data operations that can safely and gainfully tolerate errors.

Control and memory divergence between threads within the same execution bundle, or warp, have been shown to cause significant performance bottlenecks for GPU applications. In this chapter, we exploit the observation that many GPU applications exhibit error tolerance to propose energy-reliability tradeoffs through branch and data herding. Branch herding eliminates control divergence by forcing all threads in a warp to take the same control path. Data herding eliminates memory divergence by forcing each thread in a warp to load from the same memory block. To safely and efficiently support branch and data herding, we propose a static analysis and compiler framework to prevent exceptions when control and data errors are introduced, a profiling framework that aims to maximize performance

while maintaining acceptable output quality, and hardware optimizations to improve the performance benefits of exploiting error tolerance through branch and data herding. Our software implementation of branch herding on NVIDIA GeForce GTX 480 improves performance by up to 34% (13%, on average) for a suite of NVIDIA CUDA SDK and Parboil [84] benchmarks. Our hardware implementation of branch herding improves performance by up to 55% (30%, on average). Data herding improves performance by up to 32% (25%, on average). Observed output quality degradation is minimal for several applications that exhibit error tolerance, especially for visual computing applications.

7.1 Introduction

This chapter presents an evaluation of benefits achievable by a programmable stochastic processor when errors are exposed in the applications running on the processor. The evaluation is performed in the context of data intensive applications that execute on GPGPUs. Many of these applications tend to exhibit error tolerance, and the approaches presented in this chapter demonstrate how error tolerance can be exploited safely and in a way that maximizes the benefits of relaxing correctness. First, we provide background on GPUs, as well as an overview of the technique we employ to improve efficiency by relaxing correctness.

GPUs and similar SIMD architectures are becoming increasingly popular in the high-performance desktop, server, and scientific computing domains, especially as single-thread performance languishes. With the emergence of high-level programming models such as NVIDIA CUDA [85], ATI Stream, OpenCL [86], and Microsoft DirectCompute [87], and the corresponding general purpose GPUs (GPGPUs), focus has shifted from exclusively graphics processing applications to also supporting myriad data-parallel applications. Single instruction multiple data (SIMD) architectures are area and energy efficient for data-parallel applications, as instruction sequencing logic is shared by multiple execution units, leaving more area and power for the execution units themselves. However, the performance delivered by these architectures continues to lag the performance demands of emerging applications, as performance is often limited by the number of execution units that can fit within

the area and power budget of the chip. As such, performance optimizations for GPUs and other SIMD architectures are an active area of research.

The nature of SIMD execution requires that groups of parallel threads that execute together (warps) must execute the same instruction in lock-step. While the SIMD nature of execution allows the processor design to be relatively simple, application performance may suffer significantly whenever threads in the same warp behave differently due to control or memory divergence [85, 88]. Control divergence results in serialized execution of divergent control paths, leaving execution resources idle and throttling parallelism. Similarly, memory divergence causes a warp to stall until the longest memory request for a vector load completes before executing any dependent instructions. Recent work has shown that control and memory divergence between threads within a warp cause significant performance bottlenecks for many GPU applications [89, 90].

In this chapter, we propose techniques that attempt to reduce the amount of control and memory divergence in GPU applications to improve their performance. We draw on the observation that many GPU applications produce acceptable outputs even if a small number of threads in a SIMD execution unit are forced to go down the wrong control path or are forced to load from an incorrect (albeit spatially local) address. This is not surprising, considering that many GPU applications are data-intensive – different threads in a warp are often operating on similar, often spatially correlated, data. Similarly, the fraction of branches that diverge tends to be small (even though the corresponding performance degradation is large). We exploit these observations to propose two novel optimizations – *branch herding* and *data herding*. Branch herding eliminates control divergence by forcing all threads in a warp to take the same control path. This prevents serialization of branch paths that causes execution resources to remain idle for threads on the inactive control path. Data herding eliminates memory divergence by forcing each thread in a warp to load from the same memory block. This reduces the number of memory stalls and also reduces bandwidth pressure, as fewer blocks need to be loaded from memory. With the aid of static and profiling-based analyses, branch and data herding are applied discriminately to safely increase performance while maintaining acceptable output quality.

This chapter makes the following contributions:

- We demonstrate the potential for significant performance benefits without a significant degradation in output quality from carefully reducing control and memory divergence for several GPU applications that exhibit error tolerance. We confirm that an indiscriminate elimination of divergence can cause significant degradation in output quality. Similarly, a naïve implementation of divergence reduction can actually *degrade* performance in some scenarios.

- We propose two optimizations – branch herding and data herding – that eliminate control and memory divergence, respectively. Our software implementation of branch herding involves using CUDA intrinsics to force diverging threads to take the same direction at a branch as the majority of the threads. A hardware implementation of branch herding uses majority logic to identify the branch direction all threads should take. Data herding is implemented in hardware by identifying the most popular memory block (that the majority of loads map to) and mapping all loads from different threads in the warp to that block.

- While it is known that several data-parallel application can tolerate errors [91, 92, 93], what is really needed is a way to exploit available error tolerance safely and efficiently. To support our branch and data herding implementations, we also propose a static analysis and compiler framework that guarantees that control and memory errors introduced by herding will not cause exceptions, a profiling framework that aims to improve performance while maintaining acceptable output quality, and hardware optimizations to improve the performance benefits of herding.

- We quantify the potential performance benefits from different implementations of branch and data herding. Our software implementation of branch herding on NVIDIA GeForce GTX 480 improves performance by up to 34% (13%, on average) for a suite of NVIDIA CUDA SDK and Parboil [84] benchmarks. Our hardware implementation of branch herding improves performance by up to 55% (30%, on average). Data herding improves performance by up to 32% (25%, on average).

- We also evaluate output quality degradation for different GPU kernels and full applications utilizing our implementations of branch and data

herding. We provide quantitative evaluations for all applications and visual evaluations when available. Our framework aims to maintain acceptable output quality degradation for applications that can tolerate errors.

Note that our evaluations in this chapter assume a GPU architecture that matches current-generation NVIDIA CUDA devices [94, 85, 88], though we expect the ideas to be applicable to other GPU and SIMD architectures as well.

The rest of the chapter is organized as follows. Section 7.2 provides background on control and memory divergence and motivates data and branch herding. Section 7.3 describes branch herding and its various implementations. Section 7.4 describes data herding and its implementation. Section 7.5 describes a safety, performance, and output quality assurance framework for branch and data herding. Section 7.6 discusses the methodology of our study. Section 7.7 presents results and analysis. Section 7.8 discusses related work. Section 7.9 summarizes and concludes.

7.2 Background and Motivation

Below we describe the control and the memory divergence problem and discuss how carefully eliminating divergence may lead to significant performance benefits.

7.2.1 Control Divergence

SIMD architectures bolster throughput by sacrificing per-thread control flow logic in order to increase the number of execution units on a chip. Since multiple threads (a warp) execute the same instruction in lockstep on a SIMD multiprocessor (SM), only one block of instruction fetch, decode, and issue logic is needed per SM, allowing a greater fraction of the GPU's power and area budget to be spent on execution units. While such an architectural organization is beneficial for most data-parallel applications, the requirement that all threads in a warp must execute in lockstep can lead to inefficiencies when different threads take different control paths at a branch (control divergence).

```
while (--i && (xx + yy < T(4.0))) {
    y = x * y * T(2.0) + yC;
    x = xx - yy + xC;
    yy = y * y;
    xx = x * x;
} return i;
```

Figure 7.1: The main computation loop for Mandelbrot. The loop is unrolled 20 times in the actual application kernel.

Because instruction sequencing logic is shared by all execution lanes in a SM, the common mechanism for resolving control divergence in a GPU is to execute instructions from one control path for a subset of threads until reaching a point where control reconverges, then beginning execution on the other control path for the remaining threads until revisiting the reconvergence point [85, 90, 95, 96]. Since divergent branches necessarily throttle the parallelism and throughput of a SM, they can cause significant performance degradation for GPU applications [89, 90]. For a warp size of 32 (common in NVIDIA CUDA GPUs [85]), execution could be slowed down by a factor of 32 if all threads take divergent control paths through a section of code.

To understand this better, consider Mandelbrot [97] – an application from the NVIDIA CUDA SDK that exhibits control divergence. Mandelbrot generates the Mandelbrot and Julia sets – complex fractal patterns that are characterized by simple equations. Figure 7.1 shows the main loop of the kernel used to compute the Mandelbrot and Julia sets. In the actual kernel code, the loop is unrolled 20 times. Each thread in the program computes whether a particular point in the complex plane is in the Mandelbrot (or Julia) set. The program outputs images depicting the Mandelbrot and Julia sets (Figure 7.2). The color of a pixel corresponds to the number of main loop iterations (i) a thread executes to determine whether the point is in the Mandelbrot (or Julia) set.

Control divergence arises in Mandelbrot because the number of iterations required to determine whether a point is in the Mandelbrot (or Julia) set varies based on the point's location, especially in image regions near the set boundary, where some threads execute many iterations while others finish quickly. Divergence results in reduced parallelism, as some lanes in the SMs go unused while threads that have finished their computations wait until all

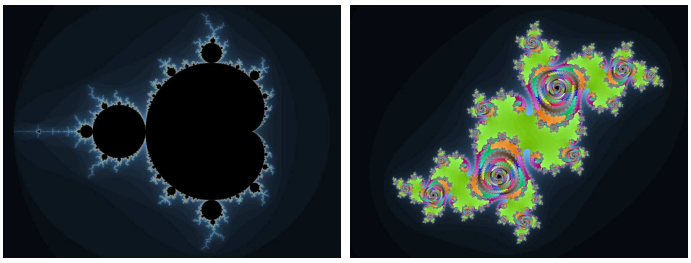


Figure 7.2: Original Mandelbrot (left) and Julia (right) images. The color of each pixel corresponds to the number of main loop iterations executed by a thread to determine whether the point is in the Mandelbrot (or Julia) set.

threads in the same warp reach a reconvergence point.

The effect of control divergence on performance can be significant. Figure 7.3 shows the potential performance increase (runtime reduction) if control divergence can be eliminated for a fraction of the static branches in Mandelbrot (from 0% to 100% of branches). The branches are chosen uniformly randomly when the fraction is less than 100%. Control divergence is preempted by changing the source code to vote within a warp on the condition of a branch and forcing all threads in the warp to take the same (majority) direction at the branch (details in Section 7.3). Experiments were run natively on a NVIDIA GeForce GTX 480 GPU (details in Section 7.6).

While only 10% of dynamic instructions in Mandelbrot are branches, and less than 1% of branches diverge, performance can potentially be increased by 31% by eliminating control divergence. As the *no software overhead* performance series in Figure 7.3 demonstrates, performance increases for Mandelbrot as control divergence is eliminated for more branches. Figure 7.4 shows that the quality of the Mandelbrot output set degrades by less than 2%, even when divergence has been eliminated for all static branches. This shows that for certain error-tolerant applications, it may be possible to get significant performance benefits from eliminating control divergence for minimal output quality degradation. A quick look at the Julia output set, however, also suggests that an indiscriminate selection of branches for herding may result in significant output quality degradation for several applications. Therefore, any implementation of branch herding should carefully select the branches to target. Figure 7.5 shows visual representations of the Mandelbrot and Julia output sets as the percentage of forced uniform branches increases from 20% to 100% in increments of 40%.

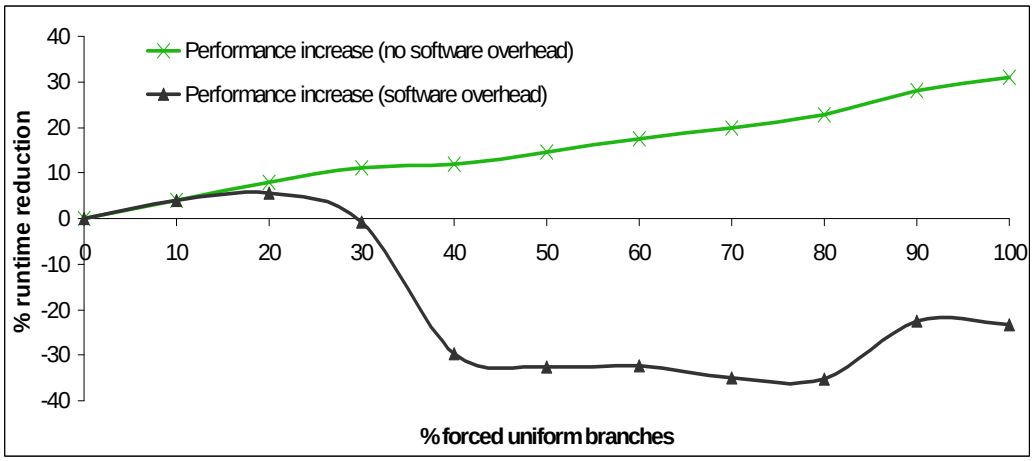


Figure 7.3: The performance of Mandelbrot can be increased by forcing uniformity for more branches. However, if software overhead is added to ensure branch uniformity, increasing the number of affected branches increases overhead and can even result in degraded performance.

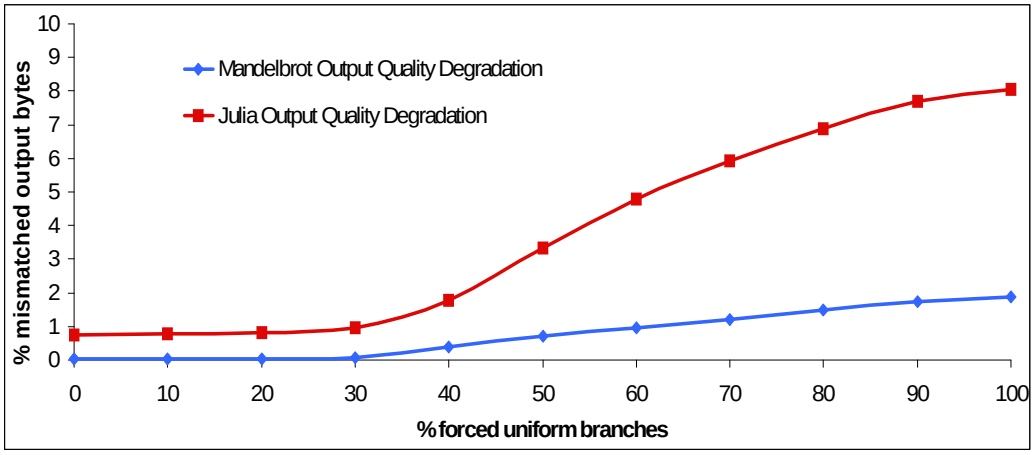


Figure 7.4: While eliminating control divergence can increase performance, blindly forcing branch uniformity can result in degraded output quality.

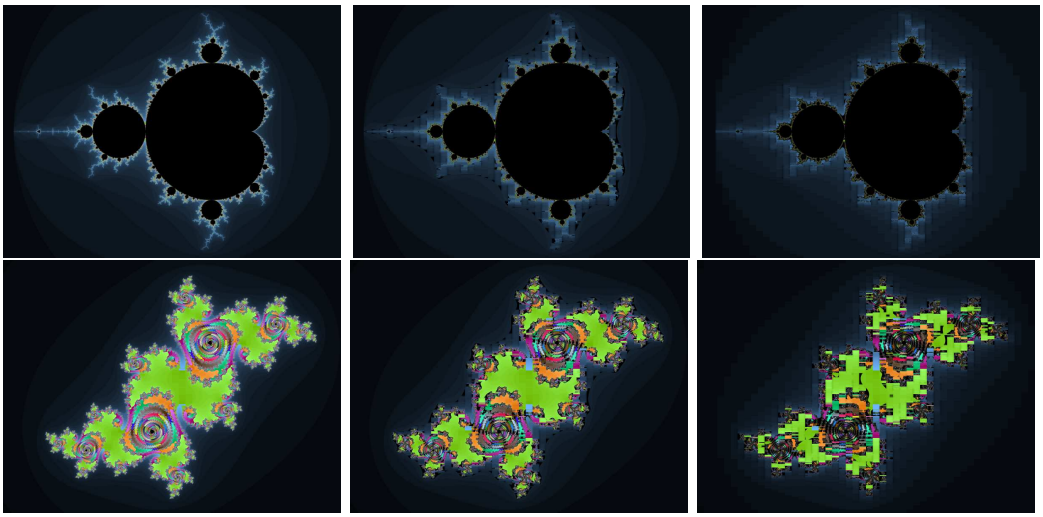


Figure 7.5: Progression of Mandelbrot (top) and Julia (bottom) images from 20% to 100% forced branch uniformity in 40% intervals.

The *software overhead* performance series of Figure 7.3 demonstrates another important consideration for any technique that eliminates control divergence. Since the fraction of divergent branches in a program may be small (in this case, less than 1%), an indiscriminate application of a technique to all branches may result in significant overhead that diminishes or even eliminates performance gains that result from reduced divergence. This result reinforces the conclusion that care should be exercised in selecting the branches to target for elimination of control divergence. Also, a low-overhead mechanism for eliminating control divergence may enable significantly more benefits. The result also confirms that naïve implementations of techniques to eliminate control divergence may actually *decrease* performance in some scenarios.

7.2.2 Memory Divergence

Like control divergence, memory divergence occurs when threads in the same warp exhibit different behavior. In the GPU, a load operation for a warp is implemented as a collection of scalar loads, where each thread potentially loads from a different address. When a load is issued, the SM sets up destination registers and corresponding scoreboard entries for each thread in the warp. The load then exits the pipeline, potentially before any of the individ-

ual thread loads have finished. When all the memory requests corresponding to the warp load have finished, the destination vector register is marked as ready. Instructions that depend on the load must stall if any lanes of the destination vector register are not ready.

Memory divergence occurs when the memory requests for some threads finish before those of other threads in the same warp [89]. Individual threads that delay in finishing their loads prevent the SM from issuing any dependent instructions from that warp, even though other threads are ready to execute. Memory divergence may occur for two reasons. (1) The time to complete each memory request depends on several factors, including which DRAM bank the target memory resides in, contention in the interconnect network, and availability of resources (such as MSHRs) in the memory controller. (2) Since the target data for a collection of memory requests made by a warp may reside in different levels of the memory hierarchy, the individual memory operations may complete in different lengths of time.

Most GPU architectures do not implement out-of-order execution due to its relative complexity and hardware cost. Rather, GPUs cover long latency stalls by multithreading instructions from a pool of ready warps. Providing each SM with plenty of ready warps ensures that long latency stalls will not be exposed. Memory divergence delays the time when a warp may execute the next dependent instruction, cutting into the pool of ready warps and potentially exposing stalls that throttle performance. Divergent memory accesses may also throttle performance by consuming additional resources, such as MSHRs and memory bandwidth. Therefore, eliminating memory divergence can potentially increase performance, especially for data-intensive GPU applications.

Another rarely discussed impact of memory divergence is on memory utilization. If different loads fetch from different memory blocks, more memory blocks need to be brought into the chip. (A memory block is the unit of memory pulled in from the memory system by a memory request.) More requests increase the bandwidth pressure on the GPU, which is often already bandwidth-limited. So, if memory divergence is eliminated (for example, when all loads fetch from the same memory block), bandwidth pressure reduces.

To gauge the potential benefit of eliminating memory divergence, we look at the SobelFilter application from the NVIDIA CUDA SDK. SobelFilter

applies an edge detection filter kernel to an input image and produces an output image. Each thread in SobelFilter loads a block of pixels from the input image and processes them in different arrangements with the edge detection kernel. We eliminate load divergence for the three kernels of SobelFilter by modifying the application so that for each load, all threads in a warp load data from the same address (that of the first active thread in the warp). Thus, the individual thread loads can be coalesced into a single memory request, making divergence impossible.

While the actual loads for individual threads in a warp may indeed diverge, the threads all load data from a localized region of the input image. Since the image data exhibits spatial correlation, eliminating divergence by loading from an address that corresponds to a neighboring pixel may often return a similar or even identical value. Figure 7.6 shows the impact on performance and output quality of increasing the fraction of warp loads that are forced to load from the same address. The figure reveals that eliminating memory divergence (forcing load uniformity) increases performance by up to 15%. However, output quality is also degraded, resulting in up to 40% mismatching bytes in the output image. Thus, some intelligence may be required to determine how and for which loads to eliminate memory divergence such that acceptable output quality is maintained. Figure 7.7 shows the Lena input image along with the pristine filter output (0% forces load uniformity), while Figure 7.8 shows a progression of output images produced by filtering the Lena input image with an increasing fraction of forced uniform loads (from 20% to 100% load uniformity).

7.3 Branch Herding

The previous section demonstrated that for an application with divergent branches, eliminating control divergence has the potential to increase performance, possibly at the expense of output quality. Due to the unique handling of divergent control flow instructions in GPUs and the forgiving nature of many data-intensive GPU applications, we propose a SIMD-specific technique for eliminating control divergence. We call our technique *branch herding*. Branch herding eliminates control divergence by herding all the threads in a warp onto the control path taken by the majority of threads.

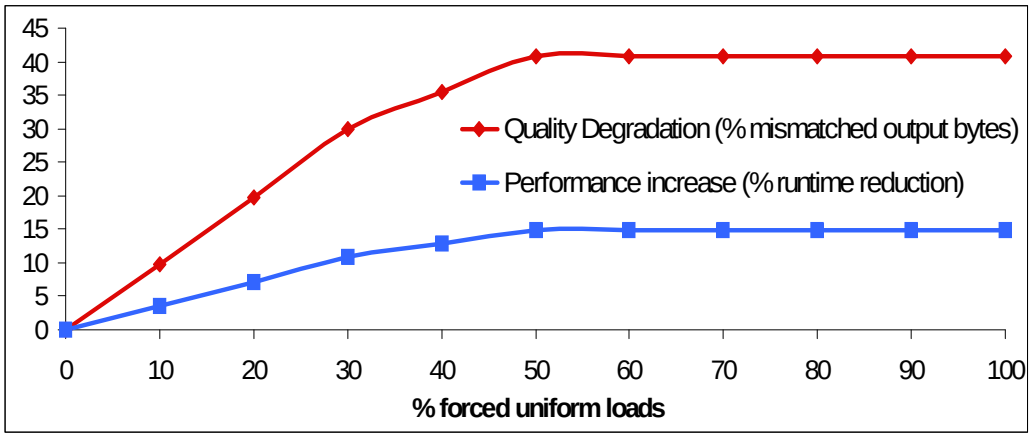


Figure 7.6: Eliminating memory divergence (forcing more uniform loads) increases performance but also degrades output quality.

Figure 7.7: Original Lena image and pristine Sobel filter output.

Figure 7.8: Lena images processed by the Sobel edge detection kernel – progression from 20% to 100% forced load uniformity in 40% intervals.

Thus, when the threads in a warp each evaluate the boolean condition for a branch, the majority outcome is decided and all threads follow the majority decision, precluding the possibility of control divergence. Because control divergence is eliminated, branch herding has the potential to increase performance for applications with divergent branches. Also, for GPU applications that can tolerate errors, acceptable output quality can be maintained when branch herding is used (see Sections 7.5 and 7.7), even though some minority of threads are allowed to perform inexact computations.

The implementation of branching in GPUs leads to benefits for branch herding in addition to the elimination of branch path serialization. The normal implementation of branching in the GPU uses a reconvergence stack and a special reconvergence instruction that is inserted before a potentially divergent branch [90, 95, 96]. The reconvergence instruction passes to the hardware the location (PC) of the reconvergence point of the branch (the next instruction that will be executed by threads on both control paths). The instruction at the reconvergence point is also flagged using a special field in the instruction encoding [95, 96]. Whenever a branch is reached, a 32-bit thread mask is computed for the warp, indicating which active threads take the branch. If the branch diverges, the mask is pushed onto the reconvergence stack, along with the PC indicating the alternate branch target and the reconvergence PC. A subset of the threads (indicated by the mask) execute the taken branch path [95, 96], while the other lanes in the SM are idle. When execution reaches the reconvergence point, the stack is popped, and the remaining threads (indicated by the mask) begin executing from the stored PC. The next time the reconvergence point is reached, all threads that originally reached the branch begin executing together again. Note that this mechanism can also handle nested divergence.

Since branch herding eliminates control divergence, the reconvergence stack is not needed for herded branches. In addition, by ensuring that all branches will be uniform branches [88], branch herding obviates the need for the special reconvergence instruction. Thus, the compiler does not insert the reconvergence instruction when the branch herding compiler flag is set or when a kernel call or particular branch instruction is marked for branch herding. It may also be possible to eliminate the reconvergence instruction by identifying the reconvergence point using a field of the branch instruction.

```
__device__ inline bool BRH(int condition){
   return (__popc(__ballot(condition)) > BRH_THRESH);
}

if( BRH(condition) )
while( BRH(condition) )
for(initial; BRH(condition); update)
```

Figure 7.9: Software branch herding implementation and example uses.

7.3.1 Software Branch Herding

Branch herding can be implemented relatively efficiently in software, using the CUDA intrinsics ballot (*__ballot*) and population count (*__popc*) [85]. The ballot intrinsic is a warp vote function that combines predicates computed by each thread in a warp and sets the N^{th} bit in a 32-bit integer if the predicate evaluates to non-zero for the N^{th} thread in the warp. In the context of branch herding, the result is a 32-bit integer that specifies the branch condition outcome for each thread in a warp. The ballot result is broadcast to a destination register for each thread in the warp. We use the population count intrinsic to count the number of set bits in the ballot result. In context, this means that each thread knows how many threads in the warp should take the branch. The branch herding function compares the population count to 16 (half warp size) and returns *true* if the majority of threads take the branch and *false* otherwise. Figure 7.9 shows the software implementation of branch herding, and provides examples of how software branch herding can be used in programs, simply by passing the condition of a control statement (e.g., if, while, for) to the branch herding procedure.

7.3.2 Hardware Branch Herding

Though our implementation of software branch herding only adds three extra instructions per branch, even this overhead may be intolerable in several scenarios, especially in tight loops or for programs that have a large fraction of branches that diverge only infrequently. Profiling information for benchmarks from the NVIDIA CUDA SDK and Parboil [84] suites that exhibit control divergence (Figure 7.10) reveals that the fraction of dynamic

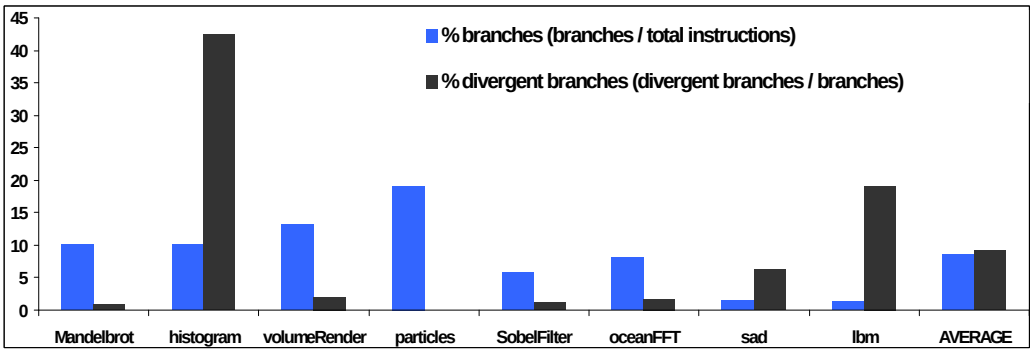


Figure 7.10: Branch statistics for applications that exhibit control divergence.

branches that diverge is indeed often very low. This is primarily because GPU programmers usually take pains to reduce potential control divergence. Nevertheless, as demonstrated in Section 7.2, even a small fraction of divergent branches can significantly reduce performance. Ideally, branch herding should be implemented as a lightweight hardware mechanism to maximize potential benefits.

For the normal implementation of branching in the SM, each active thread evaluates the branch condition to identify whether it should fetch the next instruction from the branch target or fall through. After the branch condition has been evaluated for each thread, the SM combines the condition bits from the threads to form the 32-bit branch mask and then checks for uniformity of the mask (all 0s or all 1s). If the branch is not uniform, the SM updates the reconvergence stack, as explained above.

Hardware branch herding works the same way as the normal branching implementation, but instead of evaluating the uniformity of the mask and potentially updating the reconvergence stack, the SM evaluates the majority value for the mask. The majority condition determines the next instruction for all threads in the warp. Evaluation of the majority logic can take place in the timing slack apportioned for the uniformity logic and updating the reconvergence stack (since divergence is impossible with branch herding). Thus, hardware branch herding should not affect cycle time and should not incur additional cycles of overhead. Overhead will be in terms of area, since one block of majority logic is needed per SM. However, the area of one majority block for a 32-bit word is insignificant compared with the area of the SM. Branch herding logic can be activated at a coarse granularity by

setting an enable bit in the hardware when the GPU is initialized for a kernel call or at a fine granularity by using a special field in the branch instruction to denote that the branch should be herded. The branch instruction contains an optional field (*.uni*) to identify a uniform branch (i.e., a branch for which it is possible to statically determine that the branch will not diverge). For branch herding, we override the field with a different code (*.hrd*) to indicate that the branch should be herded.

7.4 Data Herding

As discussed in Section 7.2.2, memory divergence can occur when a load instruction for a warp generates multiple memory requests that access different regions of memory or different levels of the memory hierarchy. The number of memory requests generated by a load instruction is determined by coalescing hardware in the SM [85]. Memory coalescing is performed to determine the minimum number of unique memory requests that can satisfy the individual scalar loads that make up a vector load instruction. Each scalar load address maps to a block of memory (32, 64, or 128 bytes depending on the data type), and each memory request fetches one block from memory. If multiple scalar loads map to the same block of memory, they are coalesced into a single memory request. The GPU hardware is designed such that if all scalar loads in the same warp access consecutive addresses, they can be coalesced into a single request. Besides generating memory divergence, non-coalesced loads are inefficient because they generate multiple memory requests and fetch data that is not used, wasting precious memory bandwidth and consuming additional memory controller resources such as MSHRs.

We propose a data herding implementation based on a modified coalescing policy. The modified coalescing hardware generates only one memory request for a collection of scalar loads. Rather than forming a queue of unique memory requests required to satisfy the scalar loads, the modified hardware identifies the most popular memory block (that the majority of loads map to) and maps all loads to that block – some naturally and some forcefully. This is done by comparing the number of loads that coalesce into each potential memory request and discarding requests for all but the most popular block. The upper $N - log_2(line\_size)$ bits of an N-bit address identify the memory

block that an address maps to. For any address that does not already map to the most popular memory block, the most significant $N - log_2(line\_size)$ bits of the address are overwritten with the bits that identify the most popular block. We propose data herding only for loads to ensure that all expected locations are initialized in the case of a store and to avoid conflicts that might result if stores were forcefully mapped to the same memory block.

Since our implementation of data herding ensures a single memory request for each load, and a single request is satisfied at only one level of the memory hierarchy, we prevent both types of memory divergence and also reduce memory traffic. Thus, bandwidth-limited applications may benefit substantially from data herding. Also, it is interesting to note that data herding, in itself, will never generate a memory exception, due to the nature of GPU memory design and allocation properties. In short, the threads involved all belong to the same process, and the entire memory block they will map to also belongs to the same process. An exception could, however, be generated, depending on how herded data are used later in the program. We address safety concerns associated with herding in Section 7.5.

7.5 Safety, Performance, and Output Quality Assurance for Branch and Data Herding

It is well-known that several data-parallel applications exhibit error tolerance [91, 92, 93]. To efficiently exploit this error tolerance through branch or data herding, the challenges lie in (1) guaranteeing that loading the wrong data or taking the wrong branch path will not cause an exception, and (2) maximizing performance improvement while maintaining acceptable output quality. In this section, we describe a *static analysis and compiler framework* that guarantees exception-free computation by identifying branches and data that are safe for herding, and a *profiling framework* that identifies the subset of safe branches and data for which herding increases performance while maintaining acceptable output quality.

The first step in identifying safe branches and data for herding is to identify *vulnerable operations* that, if affected by an error, *might* cause exceptions. These are pointer dereference and array reference (vulnerable to Segfault), integer division (vulnerable to INT divide by zero), and branch condition

check (vulnerable to stack overflow if an error causes infinite looping or recursion). We have written a clang [98] plugin that performs safety analysis by first parsing a program into its abstract syntax tree (AST) and searching through the AST to identify vulnerable operations.

After identifying vulnerable operations, the tool generates the control and data dependence graphs from the AST and traces through them to identify the branches and data that the vulnerable operations depend on. Then, to guarantee freedom from exceptions, the tool does not allow the compiler to insert herding directives for the branches and data identified as unsafe during static analysis. Preventing herding of "unsafe" branches and data ensures that errors induced by herding will only impact output quality. While applied here in the context of branch and data herding, the static analysis and compiler framework described in this section is generally applicable for guaranteeing safety from exceptions against any faults that affect control operations and data. Fundamentally, the framework identifies the control operations and data that should be protected from faults (and those in which faults can be allowed) in order to guarantee safety from exceptions.

After identifying which branches and data can be safely herded, the next step is to identify which can be profitably herded. As noted in Section 7.2, one challenge of branch and data herding is determining which branches and data to herd so as to improve performance while maintaining acceptable output quality. While this can be done by the programmer, often with little effort (the programmer is often aware of which branches may diverge and whether or not it would be acceptable for some threads to perform inexact computations based on the associated branches or data), we also present an automated profiling-based framework for determining which safe branches and data may be most profitable for herding.

We use the CUDA Compute Profiler [85] to determine which safe branches and which loads to safe data exhibit divergence. These branches/loads are candidates for branch/data herding. Our profiling framework starts with no herded branches/loads, progressively marks a larger fraction of the candidate branches/loads for herding, and at each step profiles the program for a set of test inputs to characterize the space of output quality degradation and performance vs. number of herded branches/loads. From this sampling we can determine an approximate upper bound on output quality degradation corresponding to a given amount of herding by selecting the worst-case degra-

```
// Static Safety Analysis
Generate Abstract Syntax Tree AST(A) for Application A
Search AST(A) to identify Vulnerable Operations VO
foreach(Vulnerable Operation v ∈ VO)
    Trace Control and Data Dependencies of v
     to classify Safe/Unsafe Branches/Data
// Quality and Performance-targeted Profiling
Profile A and identify Divergent, Safe Branches/Loads
 as Herding Candidates C
Herded Branches/Loads H = ∅
Baseline Output Quality Degradation,Performance = Q(∅), P(∅)
foreach(Candidate c ∈ C)
    foreach(Test Input t)
        Profile Output Quality Degradation Q(H + c, t)
         and Performance P(H + c, t)
    if(∃ t such that P(H + c, t) > P(H, t))
        Approximate Quality Degradation Bound
         B(H + c) = max[Q(H + c, t)|_t
        H += c
// Runtime Quality Monitoring
User specifies desired maximum Output Quality Degradation Q_max
Use Profiling Data to find maximum Herding Threshold T_h
 such that B(T_h) ≤ Q_max
Count Herding Instances I_h and disable herding when I_h == T_h
```

Figure 7.11: Pseudocode describing safety, performance, and output quality assurance framework for branch and data herding.

dation observed for a given amount of herded branches/loads. During runtime, performance counters [85] track the number of herded branches/loads and disable branch/data herding before the specified approximate threshold has been exceeded. To enable profiling and quality monitoring, the programmer should mark the variable in the code that represents output quality and specify the desired approximate bound on output quality degradation. Figure 7.11 presents pseudocode describing the control flow of our safety, performance, and output quality assurance framework for branch and data herding.

It should be noted that our profiling framework can only provide output quality guarantees for profiled inputs (or inputs similar to the profiled inputs). For all other inputs, we only provide an approximate upper bound

on output quality degradation. However, we observed that the approximate bound is often effective in practice. The challenge of understanding the mapping between faults and errors also complicates the task of output quality assurance, since the same amount of output quality degradation (in terms of a given metric) may result in noticeably different output error distributions. More precise quality metrics can help the profiler, and in conjunction with effective profiling routines, may also lead to better understanding of how faults map to errors. Creating more rigorous techniques for performance and output quality assurance is a subject of ongoing work.

Note that while hardware-based herding implementations can improve the performance benefit of herding (Section 7.7), software-based herding can be implemented for off-the-shelf GPUs and applications, and thus has the potential to demonstrate real, immediate benefits of exposing control and data errors in applications. In fact, our software herding results (Section 7.7) show speedups for applications running natively on NVIDIA GeForce GTX 480. Typically, we use data herding for all loads to the largest data structure of the application identified as safe for herding. Section 7.7 provides information on which branches and data were identified as safe and profitable for herding by our framework. Where possible, we aim for conservative results by using input data not characterized during profiling when capturing performance and output quality results.

7.6 Methodology

We perform experiments using two different execution environments. We run branch herding experiments natively on a CUDA system comprised of a NVIDIA GeForce GTX 480 GPU and a 2.27 GHz Intel Xeon E5520 CPU with 24 GB of memory. The NVIDIA CUDA v3.2 Toolkit and SDK are installed on the system.

Software branch herding performance and output quality are measured directly at runtime. Thus, reported benefits are for native execution on a state-of-art GPU architecture. To measure the number of cycles taken to execute a kernel that uses hardware branch herding ($total\_cycles_{HW\_branch\_herding\_kernel}$), we start with the number of cycles taken to execute the same kernel when software branch herding is used ($total\_cycles_{SW\_branch\_herding\_kernel}$) and use

CUDA Compute Profiler [85] performance counters to measure the number of instructions added by software branch herding function calls ($instruction\_count_{SW\_branch\_herding}$). We scale these instruction counts by the CPI for the corresponding kernels ($CPI_{SW\_branch\_herding\_kernel}$) and discount the total cycle count by this amount:

$$\begin{aligned} total\_cycles_{HW\_branch\_herding\_kernel} = & \; total\_cycles_{SW\_branch\_herding\_kernel} \\ & - instruction\_count_{SW\_branch\_herding} \\ & * CPI_{SW\_branch\_herding\_kernel} \end{aligned}$$

Since evaluating data herding requires changing the behavior of coalescing hardware and cannot be easily emulated in software, we use the GPGPU-Sim [99] simulator for our experiments. The simulator models the behavior of a NVIDIA Quadro FX 5800 GPU and can run natively-compiled CUDA v2.1 binaries.

Potentially error-tolerant benchmarks are selected from the NVIDIA CUDA SDK and Parboil [84] benchmark suites. For evaluation of branch herding, we use all benchmarks for which more than 0.5% of the dynamic branches diverge. For data herding, we select benchmarks only from the NVIDIA CUDA SDK (v2.1) that are compatible with GPGPU-Sim v2.x, which is designed around CUDA v2.1. In addition to computation kernels, we evaluate full, end-to-end benchmarks (e.g., volumeRender, particles, oceanFFT, lbm, etc.) that contain multiple kernel calls, as a partial means of demonstrating that outputs from kernels that use herding are still acceptable in the context of the greater application. Table 7.1 provides short descriptions of the benchmarks used in our evaluations.

Although we do not expect any performance overhead for hardware branch herding (Section 7.3), we collect results assuming different cycle overheads to provide both conservative and expected performance results. While we also expect that data herding based on modified coalescing can be performed in the same timing slack used for normal coalescing, we assume a cycle overhead for a more conservative estimate of the performance benefits.

Table 7.1: Benchmarks

Benchmark	Description (†CUDA SDK, ‡ Parboil)
Mandelbrot	Compute Mandelbrot and Julia sets†
histogram	64- and 256-bin Histograms†
volumeRender	Volume Rendering of 3D Textures†
particles	Particle Interaction Simulation†
SobelFilter	Sobel Edge Detection Filter†
oceanFFT	Ocean Heightfield Simulation†
binomialOptions	Binomial Option Pricing†
nbody	Gravitational n-body Simulation†
dxtc	DirectX Texture Compression†
recursiveGaussian	Recursive Gaussian Blur Filter†
lbm	Lattice-Boltzmann Method Fluid Dynamics‡
sad	Sum of Absolute Differences‡

7.7 Experimental Results

7.7.1 Branch Herding

Branch herding increases the performance of GPU applications that normally exhibit control divergence by preventing the serialization of branch paths and eliminating overheads associated with divergent branch handling. Figure 7.12 shows potential performance gains for branch herding for applications that normally exhibit control divergence. Hardware branch herding increases performance by 30% on average and up to 55% for individual applications. While we do not expect any performance overhead for hardware branch herding (see Section 7.3), we also show conservative results that assume a one-cycle overhead for hardware branch herding. Our software branch herding implementation, which runs natively on commercial GPU products, achieves 13% performance benefits, on average. Recall that the software branch herding implementation targets only safe branches that exhibit divergence *and* show benefits from software branch herding. Therefore, performance improvements are significantly higher than any naïve software branch herding implementation that targets all static branches (Figure 7.3).

Since branch herding exploits error tolerance to eliminate divergence, it may result in output quality degradation. Table 7.2 compares output quality degradation for the benchmarks with and without branch herding. Quantifying output quality degradation is difficult, because it is the consumer of the

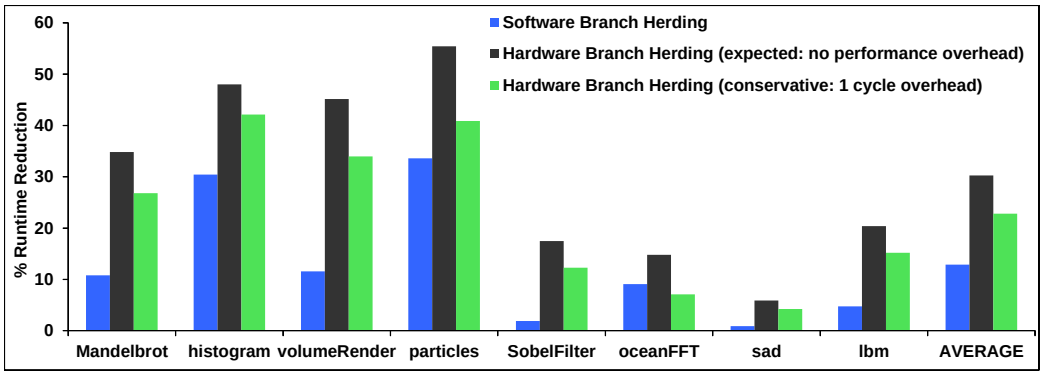


Figure 7.12: Potential performance improvement for software and hardware branch herding. Although we do not expect any additional performance overhead for our implementation of hardware branch herding, we also show a conservative performance measurement assuming a one-cycle overhead. Overhead is at most one cycle, since the additional logic (majority) is simpler than population count logic, which evaluates within a single cycle.

Table 7.2: Output quality degradation (%) for branch herding compared to original

% Mismatch	Mandelbrot	histogram	volumeRender	particles
Original	0.03	0.00	6.72	18.24
Branch Herding	1.87	5.82	7.61	18.24
% Mismatch	SobelFilter	oceanFFT	sad	lbm
Original	0.00	0.03	0.00	6.7E-7
Branch Herding	6.00	0.03	0.42	5.6E-5

data who really determines whether or not it is acceptable, and acceptability is often application-dependent. We provide output quality measurements in terms of the quality metrics incorporated by the original benchmark writers, however, our framework is modular and can easily use any other metrics (e.g., SNR) of interest to the programmer or end user. Output quality degradation is reported in terms of the fraction of mismatching bytes in the program output, except where otherwise noted. Overall, branch herding does not result in much additional output quality degradation (and degradation can be approximately bounded by our framework). Branch and data herding may be especially applicable for visual computing applications (e.g., video rendering or gaming), where performance and energy efficiency may be more critical than perfect output quality. We provide image outputs for several visual computing applications to demonstrate that post-herding output quality may often be acceptable for such applications.

Mandelbrot: In Mandelbrot, which is described in detail in Section 7.2, typically only a small fraction of dynamic branches diverge, but divergence is spread over all of the static branches in the program. Analysis identifies all branches as safe for herding. While herding more divergent branches improves performance, the amount of branch herding that can be allowed depends on the desired output quality and the region of interest in the image, since the amount of divergence depends on the region of the Mandelbrot set being viewed. Regions with intricate detail can result in substantial divergence, while monochrome regions generate no divergence. Although the overall fraction of divergent branches is often small, they can significantly impact performance. Hardware branch herding achieves about 3.5× better performance improvement than the software version, since software branch herding adds overhead to many non-divergent branches in a relatively tight loop.

Output images resemble those in Section 7.2. Note that because branch herding may estimate whether a point is in the Mandelbrot set before completely finishing the calculation for that point, even though some output pixels are not colored correctly by the application, the determination of the Mandelbrot set may be correct for those points. Thus, whether or not branch herding produces acceptable results may depend on whether the output data will be used, e.g., for a visualization or as a mathematical set.

SobelFilter: Divergence is targeted in the SobelFilter kernel (described in Section 7.2) in corner cases where the computed output pixel value for one or more threads in a warp does not lie in the valid output range. Ignoring these cases with branch herding causes the affected pixel values to roll over on the opposite side of the output range, adding some noise to the output image, which can be seen in Figure 7.13. Our framework confirms the safety of herding in this case, as it only affects pixel values. Herding is not profitable for all branches, since herding branches in tight loops that rarely diverge does not improve performance. Despite noise added by herding, edges are still detected.

histogram: Histogram has the highest fraction of divergent branches of all the applications we tested and sees considerable speedups for both software and hardware branch herding. All the divergence is caused by one static branch in a frequently-called function that adds data to the sub-histogram generated by a warp. (Sub-histograms are later merged together to create

Figure 7.13: Lena image processed by Sobel edge detection kernel with branch herding. Compare to original result in Figure 7.7.

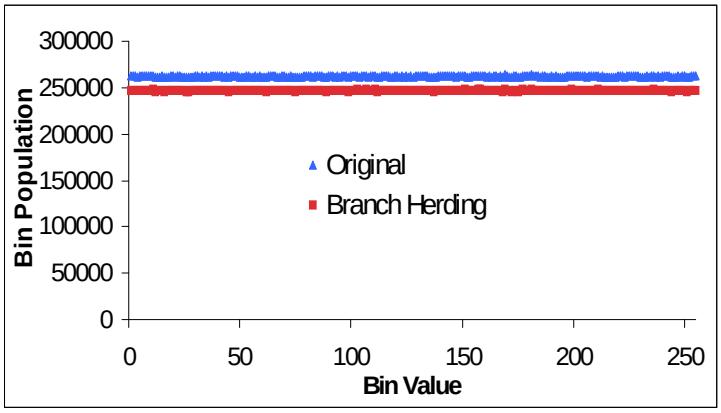


Figure 7.14: Comparison of histogram output with and without branch herding.

the final output.) This branch is safe for herding, as herding only affects histogram data. Branch herding may cause a few values not to be added to the bins, resulting in slightly undercounting the bin values. On average, bin values are undercounted by 6%, as seen in Figure 7.14. Output quality is reported as the average absolute difference between the bin values in the computed and reference outputs. It should be noted that quality degradation, and thus acceptability, depends on the characteristics of the input data.

volumeRender: VolumeRender renders a 3D texture. Although we can safely use branch herding for all the branches, most divergence is due to two static branches that cause threads to finish their computations when the object at that pixel is either opaque or too far away to be seen. Branch herding can result in some threads exiting early when the majority of threads in the same warp have finished their computations. Eliminating divergence

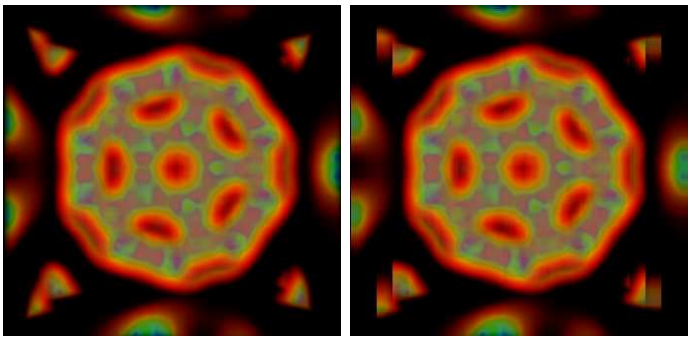


Figure 7.15: Output comparison: original volume rendering (left) and branch herding result (right).

improves performance significantly, and only increases output quality degradation by 1%. Figure 7.15 compares the original image produced by volumeRender to the image produced with branch herding.

particles: The particles application performs a simulation of physical interactions between a system of particles in an enclosed volume. The output describes the positions and velocities of the particles after a certain number of time steps. Herding branches identified by the framework only impacts these positions and velocities. A large fraction of the instructions in particles are branches that are part of collision checks between particles and with the surface of the enclosure. Even though the fraction of divergent branches is less than 1%, the number of divergent branches and the effect of divergence on performance is significant. Eliminating divergence with branch herding does not affect the output much because even if a collision is missed in one time step, it will likely be observed in a subsequent time step. The resulting collision will be slightly different, but the net effect will be similar or identical. Both software and hardware branch herding improve performance significantly without producing any noticeable degradation in the output. Whether or not results are acceptable may depend on whether the simulation is for a visualization or a scientific experiment. For example, degraded output quality may be more acceptable in a physics simulation performed for a video game.

oceanFFT: The oceanFFT benchmark computes a heightfield for a region of ocean using spectral methods. Divergence in oceanFFT arises due to boundary checks at the edge of the simulated region. Ignoring divergence with branch herding results in some slight deviations in the output around

the edges of the simulated region, but does not cause the reported output quality to change by a noticeable amount. In cases where the application would be used for a graphic visualization of the ocean, the deviations caused by branch herding would most likely be unnoticeable to the human eye.

sad: The sad benchmark performs sum of absolute differences-based motion estimation as part of the H.264 video encoder. Previous works have observed error tolerance for SAD-based motion estimation [92] due to the approximate nature of the block matching that it performs. We use branch herding for all safe branches in the sad kernel, which results in less than 0.5% output quality degradation. For most branches identified as unsafe, disallowing herding does not hurt much, since the alternate branch path is empty. In most cases, inexactness imposed by branch herding does not impact sad values enough to hinder block matching in the greater application. Thus, herding is often acceptable.

lbm: The lbm benchmark performs a lid-driven cavity fluid dynamics simulation involving a fluid that interacts with obstacles in a simulated volume. We use branch herding to eliminate divergence in the condition that tests for collisions between the fluid and an obstacle in a particular cell of the volume. Since the branch paths following the collision-detection branch contain many instructions, throughput can be affected substantially if the branch diverges. Though most cells in the volume remain error-free, branch herding causes some perturbations in the fluid simulation results. Thus, if the goal of the simulation is to simulate the fluid dynamics as accurately as possible (which may very well be the case in a scientific simulation), branch herding may be inappropriate for lbm.

7.7.2 Data Herding

Figure 7.16 shows potential for performance improvements for various benchmarks with data herding. Benefits can be substantial or nonexistent, depending on the benchmark. For the three benchmarks that do not see benefits for data herding, less than 0.2% of dynamic instructions are loads. Output quality degradation associated with data herding is compared against original output quality degradation in Table 7.3.

Data herding achieves performance benefits for two reasons. First, all

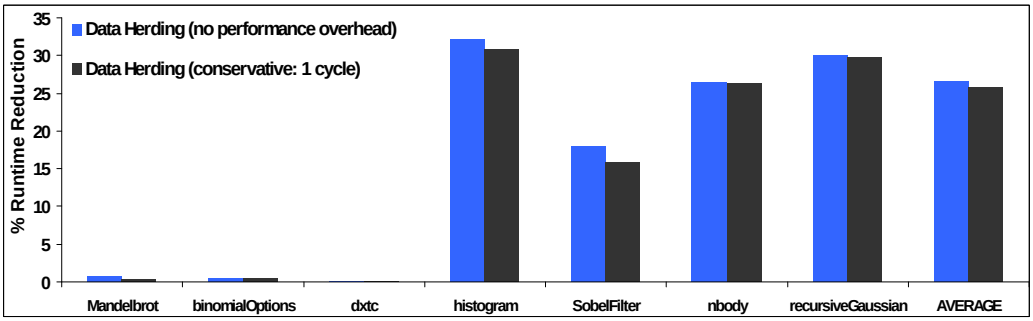


Figure 7.16: Data herding improves performance for error-tolerant benchmarks, except when the fraction of loads is very small, leaving little opportunity for improvement.

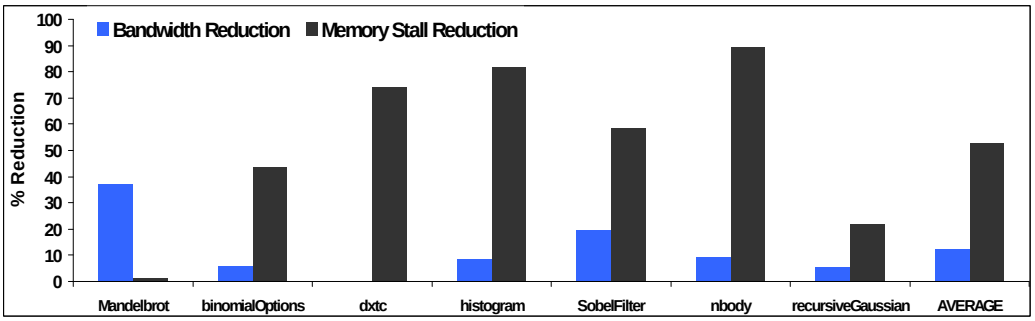


Figure 7.17: Data herding improves performance by reducing memory stalls and bandwidth usage due to divergent memory requests.

non-coalesced loads to the herded data will be coalesced into a single memory request. This reduces memory bandwidth usage and contention for resources. Reduced bandwidth and contention can also reduce the latency of memory requests. Second, since only one memory request is made for a load, memory divergence is eliminated, and warps do not spend cycles waiting for additional requests to finish after the first request returns. Figure 7.17 shows results for data herding, quantifying the reduction in bandwidth usage and in cycles that ready warps spend stalled and waiting for outstanding memory requests.

Below we explain results for individual benchmarks.

histogram: In histogram, we target loads to the initial data set to be binned in the histogram, as well as the data in the sub-histograms computed by the warps. Static analysis identifies these data as safe for herding. The benchmark consists of two kernels – one that adds values to sub-histograms and one that merges sub-histograms. Most of the speedup from data herding comes from the kernel that performs merging, since it can generate many

Table 7.3: Output quality degradation (%) for data herding compared to original

% Mismatch	Mandelbrot	histogram	nbody	binomialOptions
Original	0.02	0.00	0.00	3.8E-5
Data Herding	0.99	0.6	0.95	3.8E-5
% Mismatch	SobelFilter	dxtc	recursiveGaussian	–
Original	0.00	0.019	0.00	–
Data Herding	1.81	0.019	0.00	–

non-coalesced loads. While we observed that data herding often has only a small effect on output quality, output quality degradation depends on the characteristics of the input data. For example, uniformly distributed random data can be herded without affecting output quality substantially. On the other hand, if individual sub-histograms contain very distinct bin counts, data herding may be inappropriate for this benchmark. This brings up an important point to remember about profiling-directed herding. Output quality could potentially change undesirably for a pathological input data set. Thus, while our results do not guarantee acceptable output quality for the benchmarks over all possible data sets, they do demonstrate the potential for benefits for error-tolerant applications, especially if the target data set can be accurately characterized.

nbody: Nbody performs an all-pairs N-body simulation for a collection of bodies. The application is considerably bandwidth-limited, especially as the number of bodies increases, since the data requirement scales approximately as $O(N^2)$, stemming from the $O(N^2)$ forces that exist between N bodies. The output of the N-body simulation describes the positions of all the bodies after a specified number of timesteps. We use data herding for the body data and observe less than 1% output quality degradation, measured in terms of the average absolute difference in body positions between the computed output and a reference data set. While the deviations in the output set are visually imperceptible, they do exist. Thus, herding may be appropriate for a visualization, but may be inappropriate for a high-precision scientific simulation.

SobelFilter: As in Section 7.2, we herd image data for SobelFilter. While the performance results are similar to the maximum benefits achieved in the motivational experiment, the output quality degradation is significantly less, since loads that map to the most popular memory block receive their actual

Figure 7.18: Output comparison: original gaussian blur filtering (left) and data herding result (right).

data with our proposed implementation of data herding (Section 7.4). Output quality is also better than in the branch herding case, since data herding takes advantage of spatial correlation in the image data, which contributes to the error resilience of SobelFilter. Since the output image after herding is visually indistinguishable from the original filtered image, we omit the image here to save space and refer the reader to the images in Section 7.2.

recursiveGaussian: RecursiveGaussian performs Gaussian blur filtering on an input image. As in the case of SobelFilter, we herd the input image data. Error tolerance stems from the spatially correlated image data and the nature of the Gaussian filtering operation. Since the output value for a pixel is a weighted sum of the neighboring pixels, based on a Gaussian function, mixing in a few incorrect values is usually imperceptible, especially if the incorrect pixel values are close to the intended values due to spatial correlation. Because of the shape of the Gaussian function, the farther a neighboring pixel is from the pixel being computed, the less it affects the output. Thus, ignoring memory divergence due to non-contiguous data that cannot be coalesced usually has little effect on the output, since the data tend to be further apart in the image. We often did not observe any difference in output quality when data herding was used. Of course, output quality degradation may be greater for highly uncorrelated inputs. Figure 7.18 compares the original filter result to the result produced with data herding for one of the input images.

Mandelbrot, binomialOptions, and dxtc: For these three applications that do not see benefits from data herding, loads make up only 0.2% of the instruction mix. Thus, there is almost no potential for benefits with these

applications to begin with.

7.8 Related Work

7.8.1 Dynamic Warp Subdivision

The basic unit of SIMD execution is the warp. However, all threads in a warp must be ready in order to issue the next instruction. When SIMD restrictions stall execution, some threads in the warp may be ready while others are stalled. Normally, GPUs use warp-level multi-threading to hide latency, but this strategy requires a large, costly register file. Instead of deep warp-level multi-threading, dynamic warp subdivision [89] advocates using intra-warp latency hiding to increase throughput by allowing a divergent warp to occupy multiple scheduler slots without increasing its register usage. This scheduling approach allows threads on divergent branch paths to subdivide their warp and execute independently. Similar to a previous work advocating "diverge on miss" [100], dynamic warp subdivision also allows a subset of threads in a warp to continue execution when the remaining threads are still waiting on memory. The main drawback to dynamic warp subdivision is that it at least doubles the complexity and hardware cost of scheduling logic for each SM [89].

7.8.2 Dynamic Warp Formation

The goal of dynamic warp formation [90] is to increase hardware utilization by dynamically combining threads from multiple divergent warps. When multiple warps diverge, threads that take the same branch direction in one warp can be grouped with threads that take the same branch direction in other warps. Thus, fuller warps are formed dynamically, increasing throughput and partially mitigating the inefficiency caused by control divergence. The scheduler forms new warps out of ready threads by grouping threads that have the same next PC. Thread block compaction [101] applies dynamic warp formation whenever a divergent branch is encountered by synchronizing warps and compacting them into new warps, in which all threads take the same control path. A large warp microarchitecture [102] performs a similar optimization

by exposing a larger warp of threads to the scheduler, which is able to select SIMD width-sized sub-warps that have the same control behavior.

While dynamic warp formation has the potential to increase throughput for some applications, it is not always possible to find enough divergent threads that take the same branch direction to fill a warp within the scheduling window of available warps. Thread block compaction may help in this regard, but in some cases, warps must remain partially empty anyway, even with the additional hardware overhead required for dynamic warp formation. Nested divergence complicates the problem, making it harder to find a full warp of threads with the same next PC.

Dynamic warp formation also adds complexity in the register file, which is typically heavily banked, such that each lane of a SM can access one bank of the register file. Dynamically grouping multiple threads from the same home lane into the same warp requires adding a crossbar network so that each thread can access its registers when mapped to a different lane than its home lane. Dynamic warp formation also results in bank conflicts when multiple threads from the same home lane are grouped into the same warp, such that register file accesses are serialized over multiple cycles. One possible solution to this problem involves passing along the home lane that a thread belongs to and using lane information during dynamic warp formation so that threads are only grouped together if they belong to different home lanes. This method reduces bank conflicts, but it adds complexity to the dynamic warp formation hardware and also makes it somewhat harder to find threads that can be grouped into efficient, full warps, potentially diminishing the effectiveness of dynamic warp formation. Furthermore, for some divergence patterns, it is impossible to group threads in this manner [90].

7.8.3 Divergence Avoidance Through Software Transformation

Besides hardware-based techniques such as those discussed above, software-based techniques for avoiding divergence have also been proposed [103, 104]. These techniques aim to avoid divergence by re-mapping memory or transforming memory references to reorganize the layout of data, improve memory coalescing, and reduce control and memory divergence. Like software-based

herding, these software-based techniques have the benefit of being immediately deployable on real GPUs.

7.8.4 Energy-Reliability Tradeoffs for Error-Tolerant Applications

Related works on best-effort computing for a GPU version of semantic document search [91] and parallel implementations of recognition and mining applications [105] also recognize and exploit the forgiving nature of certain parallel algorithms to increase performance by relaxing correctness. The authors observe acceptable results for target applications after relaxing data dependencies and dropping computations. They relax data dependencies between iterations of a function call to give the parallel processor or GPU more work to do in parallel. They also monitor the usefulness of iteratively computed data during runtime and drop computations between iterations when the observed usefulness of the computed data falls below a threshold. The idea of exploiting the forgiving nature of parallel applications to improve performance is common to this dissertation. We, however, propose a different set of optimizations that target GPU- and SIMD-specific inefficiencies.

A similar work demonstrates that reliability can be traded for increased efficiency in certain data-parallel workloads [93]. The authors argue that data-parallel physics animations require perceptibility, rather than strict numerical correctness. Accordingly, they propose reducing floating point precision to improve energy efficiency. Exploiting error tolerance enables higher performance for the same cost, as they can afford to put more, reduced-precision FPUs on a chip, as opposed to fewer, high-precision FPUs.

Scalable effort hardware design [106] exploits algorithmic error tolerance in order to improve the energy efficiency of hardware. Since some algorithms are naturally tolerant to errors, scalable effort hardware design proposes to relax the traditional requirement for exact equivalence between the hardware specification and the hardware implementation. A hardware design that approximately adheres to the design specifications may provide acceptable output quality when running a robust application. The focus of scalable effort hardware design is to identify mechanisms at the circuit, architecture, and algorithm levels that influence the exactness or correctness of the final

result, and expose these mechanisms as knobs during design optimization. In this way, output quality can be traded for energy efficiency.

The authors of [106] stress the importance of cross-layer optimizations, claiming that simultaneous consideration of optimizations at all design layers results in a more efficient design than when optimizations in each layer are considered separately. This claim should easily hold true, since simultaneous consideration of more axes of optimization should prevent locally optimal, globally sub-optimal decisions. Nevertheless, in order to maintain truly "scalable" effort in the hardware design, one should take care not to overly increase the complexity of making design decisions. Whenever one considers many layers of design optimization simultaneously, the optimization space that must be evaluated to make a single decision explodes, and the complexity of making each decision grows.

Code perforation [107] also takes advantage of noise tolerance in applications to reduce energy. Proponents of code perforation argue that while some applications already trade accuracy for performance, the tradeoffs are typically application-specific, as they require algorithmic changes. Instead, code perforation focuses on program modifications that can be made automatically by a compiler to trade accuracy for performance. The idea relies on the assumption that some programs can achieve acceptable output quality even if some of the operations in the program are forgone. As such, code perforation proposes to skip non-essential lines of code in order to increase performance. By monitoring the effect of code perforation, distortion is kept within user-defined bounds. Since both output quality and performance are monitored, the code perforation compiler can either maximize performance for a given output quality or maximize output quality for a given performance target. For many applications where code perforation can be applied, perforating code would be similar to changing the size of the sample population in a statistical sampling-based problem.

7.8.5 Outcome Tolerant Branches

A work on Y-branches [108] showed that taking the wrong direction for some branches may still bring the processor to a correct architectural state. By toggling the outcome of random branches in a program, the authors observed

that for 40% of dynamic branches, taking either branch direction leads to a valid architectural state. They observed that the percentage was higher (around 50%) when allowing a mispredicted branch to continue executing on the wrong path. The authors note that outcome tolerance (the property of a branch indicating that the program output does not depend on the chosen branch direction) is a result of redundancies inserted by the programmer or compiler, as well as partially dead code.

Branch herding may benefit from outcome-tolerance in branches, but does not require it. In general, herding relies on the error-resilient nature of applications to tolerate inexactness in some thread computations. In this chapter, we also evaluate the effect on program outputs of allowing some branches to take incorrect control paths, observing acceptable outputs for many applications. In our experiments, we never observed a program crash as a result of herding branches onto the same branch path.

7.8.6 Application and Programming Language Support for Stochastic Computing

Stochastic processors enable reliability to be traded for increased energy efficiency when some form of error resilience is available. While some classes of applications often exhibit natural error tolerance (e.g., data-intensive applications), other classes of applications (or parts of applications) may not be naturally amenable to making energy-reliability tradeoffs. The static analysis framework presented in this chapter provides a means of determining where it is safe to allow errors in applications. Some related works [109, 110, 111] present application and programing language techniques that aim to manage the impact of errors on a program or make applications more suitable for execution on stochastic processors that can allow errors to propagate from hardware to software.

EnerJ [109] is a programming language extension that gives programmers more control over how programs execute on stochastic processors. EnerJ specifies how a stochastic processor is allowed to make energy-reliability tradeoffs for a given piece of code by allowing the programmer to specify which variables in the program are allowed to be approximate and which should be precise. Variables that are marked as approximate can take advan-

tage of energy-reliability tradeoffs, for example, by using unreliable memories and approximate or error-prone arithmetic units.

To maintain the dichotomy between approximate and precise program state, EnerJ performs type checking that prevents an approximate variable from affecting the assignment of a precise variable. If approximate variables need to be used in combination with precise variables, EnerJ allows an approximate variable to be "endorsed" so that it may be used in future precise computations.

Relax [110] is a framework for managing stochasticity in hardware by specifying how errors should be allowed to affect software. The potential benefits of Relax come from relaxing correctness guards imposed on hardware and allowing errors to propagate to software. The Relax framework uses an ISA extension, along with a try-catch-like software construct, that allows a programmer to specify regions of code in which correctness can be relaxed and errors can be allowed to propagate to software.

Relax relies on hardware error detection but does not support hardware error recovery, because of the relatively high cost of correcting errors in hardware. Instead, Relax performs error recovery in software. The Relax framework allows the programmer to specify an error rate for a given block of code. Relax treats all errors as equals, using a single user-specified recovery strategy such as retry or ignore when an error is detected. In the current formulation of Relax, faulty results are always discarded. Software frameworks like Relax may be useful for managing stochasticity in programs that run on programmable stochastic processors.

Some applications (such as the data intensive applications discussed in this chapter) naturally tolerate errors and can readily be executed on stochastic processors. Still, many applications cannot naturally tolerate errors. Application robustification [111] aims to transform an application into a version that is more robust to errors, to enable execution of a larger fraction of the application on a stochastic processor. In order to ensure acceptable output quality on hardware that may make errors, application robustification proposes to transform applications into numerical optimization problems that can be solved with stochastic optimization techniques. Since these optimization techniques converge to the correct result, even when computations are noisy, the robustified applications are naturally error-tolerant. Although the robust, stochastic optimization version of a program may take many itera-

tions to converge to an acceptable result, each iteration of the program has a low energy cost. In this way, it is possible to save energy over a deterministic program execution. Furthermore, as variations become more common, application robustification may become useful simply as a means of achieving acceptable results on hardware that is necessarily stochastic [111, 112].

While some applications require application robustification techniques to achieve acceptable results on stochastic hardware, other applications exhibit natural error tolerance. Algorithmic approximate correction [113] relies on the inherent error tolerance exhibited by many applications and applies approximate error correction to improve output quality, resulting in a higher quality, though potentially noisy output. The goal is to ensure that even in the presence of errors, output quality is not degraded by more than an acceptable threshold. Algorithmic approximate correction can be used in different scenarios to provide performance or output quality guarantees for an application running on a stochastic processor.

7.9 Summary

In this chapter, we demonstrate that significant potential performance benefits are possible for a programmable stochastic processor from safely and efficiently relaxing correctness and exposing errors in GPU applications. We propose two optimizations – branch herding and data herding – that relax correctness and improve performance by eliminating control and memory divergence. To ensure safety when introducing control and memory errors in applications, while targeting performance benefits and acceptable output quality, we propose a static analysis and compiler framework, a profiling framework, and hardware support for branch and data herding. Our software implementation of branch herding uses CUDA intrinsics and forces diverging threads to take the same direction at a branch as the majority of the threads. Our hardware implementation of branch herding uses majority logic to identify the branch direction all threads should take. Data herding is implemented in coalescing hardware by identifying the most popular memory block (that the majority of loads map to) and mapping all loads from different threads in the warp to that block. Our software implementation of branch herding on NVIDIA GeForce GTX 480 improves performance by up

to 34% (13%, on average) for a suite of NVIDIA CUDA SDK and Parboil [84] benchmarks. Our hardware implementation of branch herding improves performance by up to 55% (30%, on average). Data herding improves performance by up to 32% (25%, on average). For this level of performance benefits, observed output quality degradation is minimal for several applications that exhibit error tolerance, demonstrating that a programmable stochastic processor can achieve significant benefits while maintaining acceptable output quality for a large class of applications.

CHAPTER 8

SUMMARY AND FUTURE DIRECTIONS

Shrinking device sizes and growing static and dynamic non-determinism challenge the reliable manufacturing and operation of circuits. Faking determinism on inherently noisy hardware imposes significant and growing performance and power overheads. The rigid correctness contract between hardware and software leaves potential performance and energy benefits untapped, especially for applications that do not require perfect correctness and can tolerate some errors. Rather than hiding variations under expensive guardbands, stochastic processors relax traditional correctness constraints and deliberately expose hardware variability to higher levels of the compute stack, thus tapping into potentially significant performance and energy benefits, but also opening the potential for errors.

This dissertation proposes techniques for designing and architecting programmable stochastic processors and applications. Programmable stochastic processors embrace the inherent non-determinism in hardware and exploit available error tolerance in software to improve energy efficiency. Specifically, this dissertation describes design, architecture, compiler, and application optimization techniques for programmable stochastic processors and demonstrates significant benefits through detailed evaluations, including evaluations for a real processor prototype. However, going forward, there are still several issues to tackle.

Some of the main challenges lie in making stochastic computing generalizable for a wider range of applications, and doing this in a way that is automated and easy to work with. For example, Chapter 7 presents techniques for improving energy efficiency by exposing errors in applications that are naturally error-tolerant. One of the directions where more work is needed involves taking applications that are not naturally error-tolerant and transforming them into versions that are robust to errors. Section 7.8.6 discusses one promising technique for application robustification, but the technique

is limited in that it can only be applied to a limited class of applications and converting an application into its robust form is done manually. Going forward, programmable stochastic processors can benefit substantially from automated application robustification techniques that are suitable for a wider range of applications.

Current synthesis flows do not take as input user-level metrics. However, as this dissertation demonstrates, one of the primary drivers for stochastic computing is the fact that many applications can tolerate some errors. Thus, there is potential to improve the energy efficiency of synthesized designs through approximate synthesis that synthesizes a design to meet a given set of user-level constraints, including reliability. The challenge of approximate synthesis is to take an exact design specification and its reliability requirements as input and synthesize an inexact design implementation that is more efficient than an exact implementation and that meets the specified reliability requirements. One approach currently in development involves creating approximate DesignWare [114] libraries and an approximate synthesis tool that synthesizes an optimized design for a given set of user-level constraints.

At the architecture level, stochastic architecture frameworks can improve efficiency by ceasing to treat all errors as equals and considering instead the software-level criticality of errors. Compiler-level work can take advantage of dynamic compilation techniques to route instructions around errors, and static compilation techniques like those described in Chapter 7 that guarantee safety when errors are introduced into applications. In testing, yield and profit could be increased with low-overhead, fine-grained binning strategies that deem more chips useful for specific purposes. As static and dynamic nondeterminism continue to increase, and more opportunities for application-level error tolerance are discovered and created, the benefits of programmable stochastic processors will continue to grow as well.

REFERENCES

[1] ITRS, "International technology roadmap for semiconductors," 2010. [Online]. Available: http://www.itrs.net/reports.html

[2] S. Jones, "Exponential trends in the integrated circuit industry," 2008. [Online]. Available: http://www.icknowledge.com

[3] K. Bernstein, D. Frank, A. Gattiker, W. Haensch, B. Ji, S. Nassif, E. Nowak, D. Pearson, and N. Rohrer, "High-performance CMOS variability in the 65-nm regime and beyond," *IBM Journal of Research and Development*, vol. 50, no. 4, pp. 433–449, 2006.

[4] Y. Cao, P. Gupta, A. Kahng, D. Sylvester, and J. Yang, "Design sensitivities to variability: Extrapolations and assessments in nanometer VLSI," in *Proc. IEEE ASIC/SOC Conference*, 2002, pp. 411–415.

[5] P. Gupta, "Design for ultra-low-k1 patterning and manufacturing," in *Tutorial at ICMTS*, 2009.

[6] S. Das, C. Tokunaga, S. Pant, W. Ma, S. Kalaiselvan, K. Lai, D. Bull, and D. Blaauw, "Razor II: In situ error detection and correction for PVT and SER tolerance," *Proc. ISSCC*, pp. 400–622, 2008.

[7] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," in *Automata Studies*, 1956, pp. 43–98.

[8] N. Shanbhag, R. Abdallah, R. Kumar, and D. Jones, "Stochastic computation," in *Proc. DAC*, 2010, pp. 859–864.

[9] P. Mars and W. Poppelbaum, *Stochastic and Deterministic Averaging Processors*. London, UK: P. Peregrinus, 1981.

[10] J. Esch, "RASCEL, a programmable analog computer based on a regular array of stochastic computing element logic," Ph.D. dissertation, University of Illinois, Urbana, IL, 1969.

[11] N. Pippenger, "Reliable computation by formulas in the presence of noise," *IEEE Transactions Information Theory*, vol. 34, no. 2, pp. 194–197, March 1988.

[12] T. Feder, "Reliable computation by networks in the presence of noise," *IEEE Transactions Information Theory*, vol. 35, no. 3, pp. 569–572, May 1989.

[13] W. Evans and L. Schulman, "Signal propagation, with application to a lower bound on the depth of noisy formulas," in *Annual Symposium on Foundations of Computer Science*, 1993, pp. 594–603.

[14] B. Hajek and T. Weller, "On the maximum tolerable noise for reliable computation by formulas," *IEEE Transactions Information Theory*, vol. 37, no. 2, pp. 388–391, March 1991.

[15] R. Hegde and N. Shanbhag, "Energy-efficient signal processing via algorithmic noise-tolerance," in *ISLPED*, 1999, pp. 30–35.

[16] G. Varatkar, S. Narayanan, N. Shanbhag, and D. Jones, "Variation-tolerant, low-power PN-code acquisition using stochastic sensor NOC," in *ISCAS*, 2008, pp. 380–383.

[17] E. Kim and N. Shanbhag, "Soft N-modular redundancy," *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 323–336, March 2012.

[18] L. Chakrapani, B. Akgul, S. Cheemalavagu, P. Korkmaz, K. Palem, and B. Seshasayee, "Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology," *Proc. DATE*, pp. 1110–1115, 2006.

[19] D. Ernst, N. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," *Proc. IEEE/ACM MICRO*, pp. 7–18, 2003.

[20] K. Bowman, J. Tschanz, C. Wilkerson, S. Lu, T. Karnik, V. De, and S. Borkar, "Circuit techniques for dynamic variation tolerance," in *DAC*, 2009, pp. 4–7.

[21] B. Greskamp and J. Torrellas, "Paceline: Improving single-thread performance in nanoscale CMPs through core overclocking," *PACT*, pp. 213–224, 2007.

[22] S. Dhar, D. Maksimovic, and B. Kranzen, "Closed-loop adaptive voltage scaling controller for standard-cell ASICs," *IEEE/ACM ISLPED*, pp. 103–107, 2002.

[23] T. Kehl, "Hardware self-tuning and circuit performance monitoring," *IEEE International Conference on Computer Deisgn*, pp. 188–192, 1993.

[24] T. Burd, S. Member, T. Pering, A. Stratakos, and R. Brodersen, "A dynamic voltage scaled microprocessor system," *IEEE Journal of Solid-State Circuits*, vol. 11, no. 35, pp. 1571–1580, 2000.

[25] R. Sproull, I. Sutherland, and C. Molnar, "The counter-flow pipeline processor architecture," *IEEE Computer Society Press*, vol. 11, pp. 48–59, 1994.

[26] S. Herbert and D. Marculescu, "Variation-aware dynamic voltage/frequency scaling," in *15th International symposium on High-Performance Computer Architecture (HPCA'09)*, November 2009.

[27] J. Patel, "CMOS process variations: A critical operation point hypothesis," 2008. [Online]. Available: http://www.stanford.edu/class/ee380/Abstracts/080402-jhpatel.pdf

[28] J. Sartori and R. Kumar, "Alleviating voltage scaling limitations of razor-based designs," in *Proceedings of the 18th IEEE Workshop on Logic and Synthesis (IWLS 2009)*, 2009.

[29] L. Chakrapani, P. Korkmaz, B. Akgul, and K. Palem, "Probabilistic system-on-a-chip architectures," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 12, no. 3, pp. 1–28, August 2007.

[30] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *DATE*, 2010, pp. 1560–1565.

[31] S. Narayanan, J. Sartori, R. Kumar, and D. Jones, "Scalable stochastic processors," in *DATE*, 2010, pp. 335–338.

[32] A. Kahng, S. Kang, R. Kumar, and J. Sartori, "Designing processors from the ground up to allow voltage/reliability tradeoffs," in *IEEE HPCA*, 2010, pp. 119–129.

[33] A. Kahng, S. Kang, R. Kumar, and J. Sartori, "Slack redistribution for graceful degradation under voltage overscaling," in *IEEE/SIGDA ASPDAC*, 2010, pp. 825–831.

[34] S. Narayanan, G. Lyle, R. Kumar, and D. Jones, "Testing the critical operating point (COP) hypothesis using FPGA emulation of timing errors in over-scaled soft-processors," in *SELSE 5 Workshop - Silicon Errors in Logic - System Effects*, March 2009.

[35] D. Sylvester, D. Blaauw, and E. Karl, "ElastIC: An adaptive self-healing architecture for unpredictable silicon," *IEEE Design and Test of Computers*, vol. 23, no. 6, pp. 484–490, June 2006.

[36] J.-W. van de Waerdt, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J.-P. van Itegem, D. Amirtharaj, K. Kalra, P. Rodriguez, and H. van Antwerpen, "The TM3270 media-processor," in *MICRO*, Nov. 2005, pp. 12–342.

[37] "Synopsys design compiler user's manual," Synopsys. [Online]. Available: http://www.synopsys.com/

[38] "Cadence SOC encounter user's manual," Cadence. [Online]. Available: http://www.cadence.com/

[39] "Cadence SignalStorm user's manual," Cadence. [Online]. Available: http://www.cadence.com/

[40] T.-C. Chen, Y.-H. Chen, S.-F. Tsai, S.-Y. Chien, and L.-G. Chen, "Fast algorithm and architecture design of low-power integer motion estimation for H.264/AVC," *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, vol. 17, no. 5, pp. 568–577, May 2007.

[41] Joint Video Team, "JM reference software JM10.2," Online: http://iphome.hhi.de/suehring/tml/.

[42] B. Shim, S. R. Sridhara, and N. R. Shanbhag, "Reliable low-power digital signal processing via reduced precision redundancy," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, no. 5, pp. 497–510, May 2004.

[43] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge, "Opportunities and challenges for better than worst-case design," in *Asia and South Pacific Design Automation Conference*, 2005, pp. 2–7.

[44] P. Viola and M. J. Jones, "Robust real-time face detection," *International Journal of Computer Vision*, vol. 52, no. 2, pp. 137–154, 2004.

[45] P. Gupta, A. B. Kahng, and P. Sharma, "A practical transistor-level dual threshold voltage assignment methodology," in *International Symposium on Quality Electronic Design*, 2005, pp. 421–426.

[46] P. Gupta, A. B. Kahng, P. Sharma, and D. Sylvester, "Gate-length biasing for runtime-leakage control," *IEEE Trans. on Computer-Aided Design*, vol. 25, no. 8, pp. 1475–1485, 2006.

[47] "Cadence NC-verilog user's manual." Cadence. [Online]. Available: http://www.cadence.com/

[48] A. Kahng, S. Kang, R. Kumar, and J. Sartori, "Recovery-driven design: A methodology for power minimization for error tolerant processor modules," in *ACM/IEEE DAC*, 2010, pp. 825–830.

[49] D. Pisinger, "A minimal algorithm for the multiple-choice knapsack problem," *European Journal of Operational Research*, vol. 83, pp. 394–410, 1995.

[50] J. W. Tschanz, K. Bowman, S.-L. Lu, P. Aseron, M. Khellah, A. Raychowdhury, C. T. B. Geuskens, C. Wilkerson, T. Karnik, and V. De, "A 45 nm resilient and adaptive microprocessor core for dynamic variation tolerance'," in *International Solid-State Circuits Conference*, 2010, pp. 282–283.

[51] J. Sartori and R. Kumar, "Overscaling-friendly timing speculation architectures," in *ACM/IEEE GLSVLSI*, 2010, pp. 209–214.

[52] Sun, "Sun OpenSPARC project," 2010. [Online]. Available: http://www.sun.com/processors/opensparc/

[53] "Synopsys primetime user's manual," Synopsys. [Online]. Available: http://www.synopsys.com/

[54] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[55] D. M. Tullsen, "Simulation and modeling of a simultaneous multithreading processor," in *22nd Annual Computer Measurement Group Conference*, 1996, pp. 819–828.

[56] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *ISCA*, 2000, pp. 83–94.

[57] K. Yeager, "The MIPS R10000 superscalar microprocessor," in *IEEE/ACM International Symposium on Microarchitecture*, 1996, pp. 28–40.

[58] Intel Corporation, "Intel atom processor z5xx series," 2008.

[59] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program analysis," in *JILP*, 2005.

[60] B. Hargreaves, H. Hult, and S. Reda, "Within-die process variations: How accurately can they be statistically modeled?" in *ASPDAC*, 2008, pp. 524–530.

[61] L. Cheng, P. Gupta, C. Spanos, K. Qian, and L. He, "Physically justifiable die-level modeling of spatial variation in view of systematic across wafer variability," in *ACM/IEEE Design Automation Conference*, 2009, pp. 104–109.

[62] J. Sartori, J. Sloan, and R. Kumar, "Fluid NMR – Performing power/reliability tradeoffs for applications with error tolerance," in *Workshop on Power Aware Computing and Systems*, 2009.

[63] S. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas, "EVAL: Utilizing processors with variation-induced timing errors," *IEEE/ACM MICRO*, pp. 423–434, 2008.

[64] B. Greskamp, L. Wan, W. R. Karpuzcu, J. J. Cook, J. Torrellas, D. Chen, and C. Zilles, "Blueshift: Designing processors for timing speculation from the ground up," *Proc. IEEE HPCA*, pp. 213–224, 2009.

[65] L. Wan and D. Chen, "Dynatune: Circuit-level optimization for timing speculation considering dynamic path behavior," in *ICCAD*, 2009.

[66] J. Cong and K. Minkovich, "Logic synthesis for better than worst-case designs," in *VLSI-DAT*, 2009, pp. 166–169.

[67] J. P. Fishburn and A. E. Dunlop, "Tilos: A polynomial programming approach to transistor sizing," in *ACM/IEEE International Conference on Computer-Aided Design*, 1985, pp. 326–328.

[68] S. Sirichotiyakul, T. Edwards, C. Oh, R. Panda, , and D. Blaauw, "Duet: An accurate leakage estimation and optimization tool for dual-Vt circuits," *IEEE Trans. on VLSI Systems*, vol. 10, no. 2, pp. 79–90, 2002.

[69] A. Sultania, D. Sylvester, and S. S. Sapatnekar, "Tradeoffs between gate oxide leakage and delay for dual t_{ox} circuits," in *ACM/IEEE Design Automation Conference*, 2004, pp. 761–766.

[70] P. Gupta, A. B. Kahng, P. Sharma, and D. Sylvester, "Selective gate-length biasing for cost-effective runtime leakage control," in *ACM/IEEE Design Automation Conference*, 2004, pp. 327–330.

[71] T. Kehl, "Hardware self-tuning and circuit performance monitoring," *ICCD*, pp. 188–192, 1993.

[72] V. Bertacco, T. Austin, and I. Wagner, "Bug underground," University of Michigan. [Online]. Available: http://bug.eecs.umich.edu/

[73] N. Choudhary, S. Wadhavkar, T. Shah, H. Mayukh, J. Gandhi, B. Dwiel, S. Navada, H. Najaf-abadi, and E. Rotenberg, "Fabscalar: composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template," in *ISCA*, 2011, pp. 11–22.

[74] S. Palacharla, N. Jouppi, and J. Smith, "Complexity-effective superscalar processors," in *ISCA*, 1997, pp. 206–218.

[75] Y. Pan, J. Kong, S. Ozdemir, G. Memik, and S. Chung, "Selective wordline voltage boosting for caches to manage yield under process variations," in *DAC*, 2009, pp. 57–62.

[76] X. Liang and D. Brooks, "Microarchitecture parameter selection to optimize system performance under process variation," in *ICCAD*, 2006, pp. 429–436.

[77] A. Hartstein and T. Puzak, "Optimum power/performance pipeline depth," in *MICRO*, 2003, p. 117.

[78] "Cadence LC user's manual," Cadence. [Online]. Available: http://www.cadence.com/

[79] Y. Fujimura, O. Hirabayashi, T. Sasaki, A. Suzuki, A. Kawasumi, Y. Takeyama, K. Kushida, G. Fukano, A. Katayama, Y. Niki, and T. Yabe, "A configurable SRAM with constant-negative-level write buffer for low voltage operation with 0.149 μm^2 cell in 32 nm high-k/metal gate CMOS," in *ISSCC*, 2010.

[80] W. Qian and M. Riedel, "The synthesis of robust polynomial arithmetic with stochastic logic," in *DAC*, 2008, pp. 648–653.

[81] Y. Yetim, W. Jia, S. Malik, M. Martonosi, and K. Shaw, "A system-level ISA and its applications to energy-performance-reliability scheduling and scratchpad allocation," 2010. [Online]. Available: http://www.gigascale.org/pubs/2341.html

[82] G. Hoang, R. Findler, and R. Joseph, "Exploring circuit timing-aware language and compilation," in *ASPLOS*, 2011, pp. 345–356.

[83] J. Sartori and R. Kumar, "Architecting processors to allow voltage/reliability tradeoffs," in *CASES*, 2011, pp. 115–124.

[84] The IMPACT Research Group, "Parboil benchmark suite." [Online]. Available: http://impact.crhc.illinois.edu/parboil.php

[85] *NVIDIA CUDA Programming Guide, Version 3.0*, NVIDIA, 2010.

[86] *OpenCL*, Khronos Group, 2010.

[87] *GPGPU Computing Horizons*, Microsoft, 2010.

[88] *NVIDIA Compute PTX: Parallel Thread Execution*, NVIDIA, 2009.

[89] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *ISCA*, 2010, pp. 235–246.

[90] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *MICRO*, 2007, pp. 407–420.

[91] S. Byna, J. Meng, A. Raghunathan, S. Chakradhar, and S. Cadambi, "Best-effort semantic document search on GPUs," in *GPGPU*, 2010, pp. 86–93.

[92] G. Varatkar and N. Shanbhag, "Energy-efficient motion estimation using error-tolerance," in *ISLPED*, 2006, pp. 113–118.

[93] T. Yeh, P. Faloutsos, M. Ercegovac, S. Patel, and G. Reinman, "The art of deception: Adaptive precision reduction for area efficient physics acceleration," in *MICRO*, 2007, pp. 394–406.

[94] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, NVIDIA, 2009.

[95] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *ISPASS*, 2010, pp. 235–246.

[96] B. Coon, *United States Patent #7,353,369: System and Method for Managing Divergent Threads in a SIMD Architecture*, NVIDIA, 2008.

[97] Wikipedia, "Mandelbrot set," 2011. [Online]. Available: http://en.wikipedia.org/wiki/Mandelbrot_set

[98] University of Illinois, "clang: a C language family frontend for LLVM." [Online]. Available: http://clang.llvm.org

[99] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009, pp. 163–174.

[100] D. Tarjan, J. Meng, and K. Skadron, "Increasing memory miss tolerance for SIMD cores," in *SC*, 2009, pp. 22:1–22:11.

[101] W. Fung and T. Aamodt, "Thread block compaction for efficient SIMT control flow," in *HPCA*, 2011, pp. 25–36.

[102] V. Narasiman, M. Shebanow, C. Lee, R. Miftakhutdinov, O. Mutlu, and Y. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *MICRO*, 2011, pp. 308–317.

[103] S. Che, J. Sheaffer, and K. Skadron, "Dymaxion: Optimizing memory access patterns for heterogeneous systems," in *SC*, 2011, pp. 13:1–13:11.

[104] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for GPU computing," in *ASPLOS*, 2011, pp. 369–380.

[105] J. Meng, S. Chakradhar, and A. Raghunathan, "Best-effort parallel execution framework for recognition and mining applications," in *IPDPS*, 2009, pp. 1–12.

[106] V. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. Chakradhar, "Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency," in *DAC*, 2010, pp. 555–560.

[107] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *ASPLOS*, 2011, pp. 199–212.

[108] N. Wang, M. Fertig, and S. Patel, "Y-branches: When you come to a fork in the road, take it," in *PACT*, 2003, pp. 56–67.

[109] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *PLDI*, 2011, pp. 164–174.

[110] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," *ISCA*, pp. 497–508, 2010.

[111] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi, "A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance," in *DSN*, 2010, pp. 161–170.

[112] K. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computing*, vol. 33, no. 6, pp. 518–528, 1984.

[113] J. Sartori, J. Sloan, and R. Kumar, "Stochastic computing: Embracing errors in architecture and design of processors and applications," in *CASES*, 2011, pp. 135–144.

[114] "Synopsys DesignWare building block IP user's guide," Synopsys. [Online]. Available: http://www.synopsys.com/